Technische Universität Berlin

Forschungsberichte
der Fakultät IV – Elektrotechnik und Informatik

Konzeption und Implementierung einer Anwendungsumgebung für attributierte Graphtransformation basierend auf Mathematica

Jochen Adamek[1]
Betreuer: Frank Hermann

1) tronador(at)cs.tu-berlin.de
Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany

Bericht-Nr. 2009-15
ISSN 1436-9915

Inhaltsverzeichnis

Teil I.
Einleitung

1. Kurzbeschreibung

Ich präsentiere in meiner Diplomarbeit ein auf der Software Mathematica basierendes Tool zur Modelltransformation mit Hilfe theoretischer Konzepte der getypten attributierten Graphtransformation mit Doppelpushouts. Das Tool zeichnet sich durch eine hohe Effizienz der mathematischen Berechnungen und eine Kompaktheit der Implementierung aus und soll u.a. in einem Modellierungsszenario der Geschäftsprozesse der Credite Suisse Verwendung finden.

2. Abstract

In my diploma thesis I present a tool that is based on the software Mathematica. It is concerned with the model transformation with the aid of theoretical concepts of typed attributed graph transformation with double pushouts. The tool stands out by its high efficiency in the mathematical calculations and a compact implementing. It shall be used, among others, in the business process modelling scenario of Credite Suisse.

3. Motivation

Ziel dieser Arbeit ist die Entwicklung einer effizienten Graphtransformationsengine für (getypte) attributierte Graphen basierend auf der Software Mathematica. Getypte attributierte Graphen sind als Kategorie wohlfundiert und relevant vor allem für die Metamodellierung in Bereichen des Software Engineering und der visuellen Sprachen. Die Implementierung der Graphtransformationstheorie zur regelbasierten Modifikation von Graphen dient u.a. als Grundbaustein für ein theoretisches Konzept des Enterprise Engineering bei der Credit Suisse, wobei normative Aspekte der Modelle durch Graph-Constraints analysiert und die Konsistenz der Modelle durch Modellintegration und Transformation kontrolliert werden soll. Dies soll auf Basis der Implementierung mit Mathematica geschehen. Das Framework in [*SCIB*09] verdeutlicht den enormen Praxisbezug der Realisierung mit Mathematica. Das Plugin Wolfram Workbench [*WW*] zur benutzerfreundlicheren Bearbeitung des Projektes in der Eclipse-Umgebung stellt zudem weitere Funktionaliäten von Eclipse für die Programmierung mit Mathematica zur Verfügung. Eclipse ist ein weitverbreitetes Open-Source-Framework und optimal für die Verwaltung größerer Softwareentwicklungen. Denkbar ist ebenfalls eine Einbeziehung der Mathematica-Implementierung im Bereich der Entwicklung von visuellen Editoren auf Basis verschiedener Eclipse-Plugins. Ein Motivationsgrund für meine Arbeit ist desweiteren eine Grundlage für eine Implementierung von Triple-Graph-Grammatiken in Anlehnung an das in Abschnitt 8.4 kurz beschriebene Tool MOFLON [*MO*]. Eine Realisierung der Triple-Graph-Grammatiken würde Modellintegrationen erlauben.

Neben der hohen Effizienz der Mathematica-Implementierung zeichnet sich das Graphtransformationstool vor allem auch durch eine sehr strikte Anlehnung an theoretische Konzepte aus und bietet aufgrund regelmäßiger Erweiterungen und Neuerungen der Mathematica-Software ein hohes Optimierungspotential. Ein weiterer Vorteil gegenüber einer bisherigen Implementierung mit Java in AGG (siehe 8.3), einem ebenfalls an der TU entwickelten Tool für die Graphtransformation, liegt in der Kompaktheit und daraus resultierenden besseren Übersichtlichkeit durch Ausnutzung der Eigenschaften funktionaler und objektorientierter Programmiersprachen.

4. Überblick

Diese Diplomarbeit ist in 10 Kapitel unterteilt. Beginnend mit einer kurzen Schilderung der Motivation dieser Arbeit, erfolgt in Teil 5 eine Einführung in die theoretischen Grundlagen der getypten attributierten Graphtransformation. In Teil 6 wird die proprietäre Software Mathematica und deren grundlegende Struktur und Funktionalität beschrieben. Basierend auf dem theoretischen Fundament gebe ich in Kapitel 7 einen Überblick über die Implementierungsrealisierungen mit Mathematica.

Das Kapitel 8 bezieht sich einerseits auf die Einführung zweier Benchmarks für Graphtransformationstools, andererseits aber auch auf deren Umsetzung mit Mathematica. In

der abschließenden Analyse der Implementierung wird das Mathematica-Projekt evaluiert und mit anderen Tools der Graphtransformation verglichen. Insbesondere erfolgt ein direkter und ausführlicher Vergleich zur Software AGG. Im Laufe dieser Vergleichsanalyse werden einige der bedeutendsten Graphtransformationstools vorgestellt. Im letzten Kapitel 9 werden zukünftige Projekte vorgestellt, die auf der bisher existenten Implementierung aufsetzen werden. Insbesondere werde ich auf das in [*ICGT*08] thematisierte inkrementelle Patten-Matching (siehe 9.1) für Implementierungen im Bereich der Suchalgorithmen eingehen und verdeutlichen, inwieweit eine Realisierung auch für die Mathematica-Programmierung der Graphtransformationstheorie sinnvoll wäre. In Kapitel 10 erfolgt abschließend eine Zusammenfassung über die Ziele und Ergebnisse der Arbeit.

Teil II.
Hauptteil

5. Getypte attributierte Graphtransformation

In den 60er und 70er Jahren wurden die ersten Konzepte der Graphtransformation definiert. Sie finden heutzutage in verschiedenen Bereichen der Informatik Verwendung. Die algebraische Graphtransformation ist Bestandteil vor allem in theoretischen Computerwissenschaften. Anzutreffen ist sie aber auch im Bereich des Software Engineering, zudem ist die algebraische Graphtransformation im Bereich nebenläufiger und verteilter Systeme und Techniken der visuellen Sprachen und der Modelltransformation relevant. Durch Graphstrukturen mit gerichteten Kanten können Objekte und deren Beziehungen dargestellt werden. Ich werde nun die grundlegenden theoretischen Definitionen einführen, die für die (getypte) attributierte Graphtransformation relevant sind.

5.1. E-Graphen

Ein sogenannter E-Graph besteht aus zwei verschiedenen Knotenmengen, einer Knotenmenge in Bezug auf den Graphen und einer Knotenmenge in Bezug auf die Daten sowie drei verschiedenen Arten von Kanten. Die Graphkanten gewährleisten die Verbindung einzelner Knoten miteinander, zusätzlich existieren jedoch Kanten für die Knoten- und Kantenattributierung.

Definition(E-Graph und E-Graphmorphismen)
Ein E-Graph G mit $G = (V_G, V_D, E_G, E_{NA}, E_{EA}, (source_j, target_j)_{j\in\{G,NA,EA\}})$ enthält die folgenden Komponenten:

- V_G und V_D werden Graph- und Datenknoten genannt;
- E_G, E_{NA} und E_{EA} werden Graph-, Knotenattribut- und Kantenattributkanten genannt;

  und source- und target-Funktionen
- $source_G : E_G \to V_G$, $target_G : E_G \to V_G$ für Graphkanten
- $source_{NA} : E_{NA} \to V_G$, $target_{NA} : E_{NA} \to V_D$ für Knotenattributkanten und
- $source_{EA} : E_{EA} \to E_G$, $target_{EA} : E_{EA} \to V_D$ für Kantenattributkanten.

Definition(E-Graphmorphismen)
Es existieren zwei E-Graphen G^1 und G^2 mit
$G^k = (V_G^k, V_D^k, E_G^k, E_{NA}^k, E_{EA}^k, (source_j^k, target_j^k)_{j\in\{G,NA,EA\}})$ für k = 1,2. Ein E-Graph Morphismus f : $G^1 \to G^2$ ist ein Tupel(f_{V_G}, f_{V_D}, f_{E_G}, $f_{E_{NA}}$, $f_{E_{EA}}$) mit $f_{v_i} : V_i^1 \to V_i^2$ und $f_{E_j} : E_j^1 \to E_j^2$ für $i \in \{G, D\}$, $j \in \{G, NA, EA\}$ sodaß f mit allen target- und source-Funktionen konform ist, wie z.B. $f_{V_G} \circ source_G^1 = source_G^2 \circ f_{E_G}$.

Um eine kurze Zusammenfassung zu geben, kann man konstatieren, daß der essentielle Unterschied zwischen E-Graphen und herkömmlichen Graphen dahingehend besteht, daß E-Graphen Kantenattributkanten erlauben. Deren Quelle (source) ist kein Graphknoten,

sondern eine Graphkante. Der Hauptunterschied zwischen E-Graphen und beschrifteten Graphen liegt in der Tatsache, daß Knoten- und Kantenattributkanten anstatt der für die Beschriftung zuständigen Funktionen definiert werden.

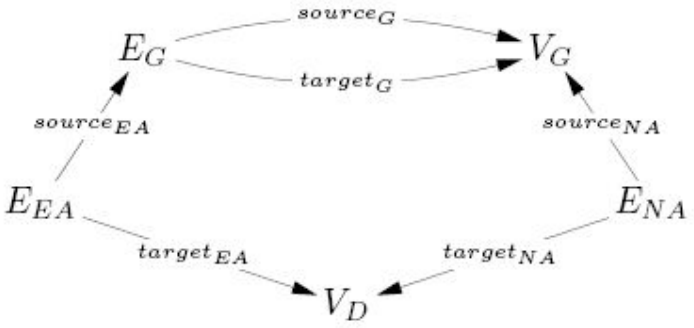

Abbildung 1: E-Graph

Definition(Kategorie EGraphs)
E-Graphen und E-Graph Morphismen bilden die Kategorie **EGraphs**.

5.2. Attributierte Graphen

Die Graphtransformation dient nicht nur als Modellierungsmethodik in der Softwaretechnik, sie wird auch für die Metasprache verwendet und im Bereich der visuellen Modellierungstechniken wie UML benutzt [*AGU*]. Graphtransformationsregeln können hier bei mehreren UML-Diagrammtypen von Bedeutung sein. Hierfür sind beschriftete Graphen nicht ausreichend, sodaß man getypte, attributierte Graphen als theoretisches Fundament wählen muss. Graphen ohne Attributierungen erlauben keine Attributberechnungen.

Ein attributierter Graph enthält eine Knotenattributierung als ein Tupel $AG = (G, A)$ eines Graphen G und einer Datentypalgebra A. Der spezielle Graph G mit seinen Knotenattributierungen und Kantenattributierungen wird gemäß [*FAGT*] als E-Graph bezeichnet. Diese Art der attributierten Graphen führt zur Definition einer Kategorie $\mathbf{AGraphs}_{ATG}$, wobei der attributierte Graph über einen attributierten Typgraphen getypt wird. Dieses Konzept findet man ebenso in der Graphtransformationsmaschine AGG (siehe 8.3) wieder.

Definition(Attributierte Graphmorphismen)
Gegeben sind zwei attributierte Graphen $AG^1 = (G^1, D^1)$ und $AG^2 = (G^2, D^2)$. Ein attributierter Graphmorphismus $f : AG^1 \to AG^2$ ist ein Paar $f = (f_G, f_D)$ aus einem E-Graphmorphismus $f_G : G^1 \to G^2$ und einem $DSIG$-Algebrahomomorphismus $f_D : D^1 \to D^2$, so daß für alle Attributwertsorten $s \in S'_D$ das Diagramm aus der Abbildung 2 kommutiert und die vertikalen Morphismen Inklusionen sind. (siehe Abbildung 2)

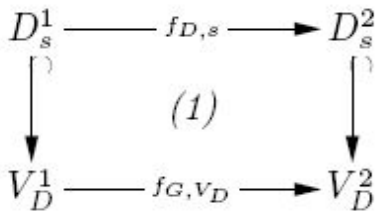

Abbildung 2: Attributierter Graphmorphismus

Definition(Kategorie AGraphs)
Die attributierten Graphen bilden mit den attributierten Graphmorphismen die Kategorie **AGraphs**.

5.3. Attributierter Typgraph

Die Typisierung der attributierten Graphen erfolgt über sogenannte Typgraphen. Alle Graphen sind mit einem Morphismus über den Typgraphen verbunden. Es folgt die Definition eines attributierten Typgraphen.

Definition(Attributierter Typgraph)
Ein attributierter Typgraph $ATG = (G, Z)$ besteht aus einem E-Graph G und einer finalen DSIG-Algebra Z. In der Regel enthalten bei den finalen Algebren die Trägermengen nur die Bezeichnung der jeweiligen Sorte. Dieser Sortenbezeichner gibt somit den Typ der Menge von Attributwerten an, die dieser Trägermenge in über ATG getypten Graphen entspricht.

5.4. Getypte attributierte Graphen

Getypte attributierte Graphen beschreiben Graphen mit einem entsprechenden Typen für Knoten und Kanten, die über einen sogenannten Typgraphen verbunden sind. Diese Verbindung wird durch einen attributierten Graphmorphismus repräsentiert. Typisierungen gewährleisten die Wahrung der Konsistenz der Modellierungen.

Definition(Getypte attributierte Graphen)
Ein über ATG getypter attributierter Graph (AG, t) besteht aus einem über $DSIG$ attributierten Graphen AG und einem attributierten Graphmorphismus $t : AG \rightarrow ATG$.

Definition(Getypte attributierte Graphmorphismen)
Gegeben sind zwei attributierte Graphen $AG^1 = (G^1, D^1)$ und $AG^2 = (G^2, D^2)$. Ein attributierter Graphmorphismus $f : AG^1 \rightarrow AG^1$ ist ein getypter attributierter Graphmorphismus $f : (AG^1, t^1) \rightarrow (AG^2, t^2)$ falls $t^2 \circ f = t^1$. Dies beschreibt die Typverträglichkeit beider Graphen.

Definition(Kategorie $\mathbf{AGraphs_{ATG}}$)
Ein getypter attributierter Graph über einem attributierten Typgraphen ATG und getypte attributierte Graphmorphismen bilden die Kategorie $\mathbf{AGraphs_{ATG}}$.

5.5. Pushout-Konstruktion

Um einen kurzen Einblick in die Definitionen der Pushout-Konstruktionen für attributierte Graphen zu geben, möchte ich kurz Pushout-Konstruktionen für Graphen einführen.

Zur effizienten Strukturierung von großen und komplexen Graphen durch Teilgraphen, die über eine disjunkte Vereinigung hinausgehen muss, wird die Pushout-Konstruktion benötigt, wobei diese Verknüpfung über einen Teilgraphen erfolgt. Anschließend werden alle anderen Knoten und Kanten der anderen beiden Graphen hinzugefügt.

Definition(Pushout)
Sind zwei Morphismen $f : A \to B$ und $g : A \to C$ in einer Kategorie C gegeben, so ist ein Pushout(D, f', g') über f und g definiert durch

- ein Pushout-Objekt D und
- einen Morphismus $f' : C \to D$ und $g' : B \to D$ mit $f' \circ g = g' \circ f$

so, daß die universelle Eigenschaft erfüllt ist: Für alle Objekte X und Morphismen $h : B \to X$ und $k : C \to X$ mit $k \circ g = h \circ f$, existiert ein eindeutiger Morphismus $x : D \to X$, so daß $x \circ g' = h$ und $x \circ f' = k$:

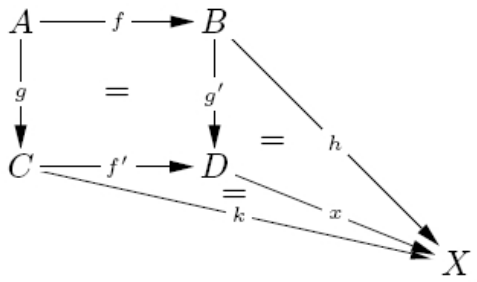

Abbildung 3: Pushout-Konstruktion

In der Regel wird die Abkürzung "PO" für "Pushout" verwendet. Man schreibt als Notation $D = B +_A C$ für das Pushout-Objekt D.

Definition(Pushouts in Sets,Graphs und $\mathbf{Graphs_{TG}}$)
In der Kategorie **S***ets* kann das Pushout-Objekt über die Morphismen $f : A \to B$ und $g : A \to C$ als Quotient $B \dot{\cup} C|_{\equiv}$konstruiert werden, wobei $\equiv$ die kleinste Äquivalenzrelation mit $(f(a), g(a)) \in \equiv$ für alle $a \in A$ ist. Die Morphismen f' und g' sind definiert durch $f'(c) = [c]$ für alle $c \in C$ und $g'(b) = [b]$ für alle $b \in B$.

Zusätzlich existieren die folgenden Eigenschaften:

1. Wenn f injektiv (oder surjektiv) ist, dann ist f' ebenfalls injektiv (oder surjektiv).

2. Das Paar (f', g') ist vereinigt surjektiv, da für jedes $x \in D$ ein Urbild $b \in B$ mit $g'(b) = x$ oder $c \in C$ mit $f'(c) = x$ existiert.

3. Wenn f injektiv ist und $x \in D$ Urbilder $b \in B$ und $c \in C$ mit $g'(b) = f'(c) = x$ hat, dann existiert ein eindeutiges Urbild $a \in A$ mit $f(a) = b$ und $g(a) = c$.

4. Wenn f und folglich auch f' injektiv sind, dann ist D isomorph zu $D' = C \dot{\cup} B \setminus f(A)$.

Die zuvor beschriebenen Definitionen für Pushout-Konstruktionen für Graphen dienen als Grundlage für Pushouts in den Kategorien $AGraphs$ und $AGraphs_{ATG}$. Näheres findet man im Lehrbuch [$FAGT$] im Kapitel 8.2. Für die Definition der getypten attributierten Graphtransformation (siehe 5.7), die auch Doppelpushout-Konstruktion bezeichnet wird, ist das Konzept der Pushouts essentiell. Die Umsetzung der Pushout-Konstruktion mit Mathematica wird in 7.5 explizit erklärt.

5.6. Pullback-Konstruktion

Die sogenannten Pullbacks sind duale Konstruktionen der Pushouts und können als generalisierte Schnittmenge der Objekte betrachtet werden, die über ein gleiches Objekt identifiziert werden. Pullbacks spielen bei mehreren in der Graphtransformation relevanten Theoremen eine essentielle Rolle. Die Pullback-Konstruktion ist kein Bestandteil der getypten attributierten Graphtransformation. Vielmehr ist das theoretische Konzept relevant für Doppelpushout-Konstruktionen mit Borrowed Context. (siehe [$BPMR$08])

Definition(Pullback)
Gegeben sind die Morphismen $f : C \to D$ und $g : B \to D$ in einer Kategorie $\mathbf{C}$, ein Pullback (A, f', g') über f und g ist definiert durch

1. ein Pullbackobjekt A und

2. Morphismen $f' : A \to B$ und $g' : A \to C$ mit $g \circ f' = f \circ g'$

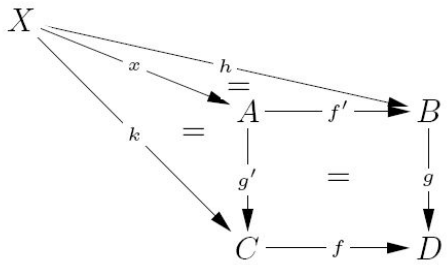

Abbildung 4: Pullback-Konstruktion

sodaß die folgende universelle Eigenschaft erfüllt ist: Für alle Objekte X mit den Morphismen $h : X \to B$ und $k : X \to C$ mit $f \circ k = g \circ h$, existiert ein eindeutiger Morphismus $x : X \to A$, sodaß $f' \circ x = h$ und $g' \circ x = k$.

Die Umsetzung der Pullback-Konstruktion mit der Software und Programmiersprache Mathematica wird in 7.6 beschrieben.

5.7. Graphtransformation

Die Theorie der Graphtransformation zur regelbasierten Änderung von Graphen basiert auf Graphproduktionen, deren Anwendung „direkte Graphtransformation" genannt wird. Gegeben sind eine Produktion $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ und ein getypter, attributierter Graph G mit einem getypten, attributierten Graphmorphismus $m : L \to G$, „match" genannt, dann wird eine direkte, getypte, attributierte Graphtransformation $G \overset{p,m}{\Rightarrow} H$ von G zu einem getypten, attributierten Graphen H bestimmt durch das folgende Doppelpushout(DPO)-Diagramm in der Kategorie $AGraphs_{ATG}$, wobei (1) und (2) Pushouts sind:

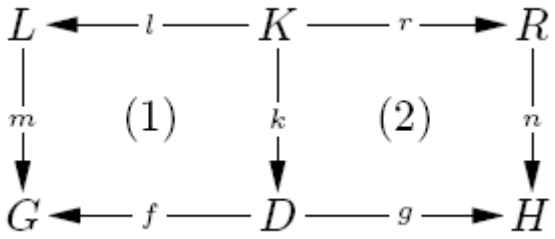

Abbildung 5: Transformation

Zwischen den einzelnen Graphen der Produktion p bestehen Morphismen, die als Inklusionen festgelegt werden. Wie Regelanwendungen in Mathematica genau aussehen, wird in Kapitel 7.10 beschrieben. Um eine Transformation durchführen zu können, muss eine Bedingung, Gluing-Condition genannt, kontrolliert werden und gültig sein. Wird die sogenannte Gluing-Condition erfüllt, existiert ein eindeutiger Kontextgraph, der synonym auch Pushout-Komplement bezeichnet wird. Da Attribute in der Umsetzung mit Mathematica lediglich als Labels betrachtet werden (siehe 7.3), möchte ich mich in dieser Arbeit auf die Definition der Gluing-Condition für Graphen beschränken. Eine ausführliche Beschreibung der Gluing-Condition für attributierte Graphtransformationen findet man in [*FAGT*] im Kapitel 9.2. Kommen wir zur Definition der Gluing-Condition für Graphen:

Definition(Gluing-Condition für Graphen)
Gegeben sind eine (getypte) Produktion in Bezug auf Graphen $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$,

ein (getypter) Graph G und ein Match $m : L \to G$ mit $X = (V_X, E_X, s_X, t_X)$ für alle $X \in \{L, K, R, G\}$, so daß die folgenden Bedingungen erfüllt sein müssen:

- Die Gluing-Points GP sind die Knoten und Kanten in L, die nicht durch p gelöscht werden, und werden wie folgt definiert: $GP = l_V(V_K) \cup l_E(E_K) = l(K)$.
- Die Identification-Points IP sind die Knoten und Kanten in L, die durch m identifiziert werden, und werden wie folgt definiert:
  $IP = \{v \in V_L \,|\exists w \in V_L, w \neq v : m_V(v) = m_V(w)\} \cup$
  $\{e \in E_L \,|\exists f \in E_L, f \neq e : m_E(e) = m_E(f)\}$.
- Die Dangling-Points DP sind die Knoten in L, dessen Ur-Bilder unter m die source oder target-Bereiche einer Kante in G, die nicht zu $m(L)$ gehört, sind. $DP = \{v \in V_L \,|\exists e \in E_G \backslash m_E(E_L) : s_G(e) = m_V(v)$ oder $t_G(e) = m_V(v)\}$.

p und m erfüllen die Gluing-Condition, wenn alle Identification-Points und alle Dangling-Points auch Gluing-Points sind, d.h. $IP \cup DP \subseteq GP$.

Sogenannte Dangling-Points müssen Teilmenge der Gluing-Points sein, da ansonsten im Graphen des Pushout-Komplements frei herumhängende Kanten existieren könnten und dies dem theoretischen Konzept von (getypten) attributierten Graphen widersprechen würde. Ein Löschen der Identification-Points würde dazu führen, daß kein Pushout-Komplement existiert. Auch muss der Morphismus l zwischen dem Klebegraphen und der linken Regelseite injektiv sein, da ansonsten die Eindeutigkeit des Pushout-Komplements nicht gewährleistet werden kann. Der Code in Mathematica bezüglich der Gluing-Condition wird in Kapitel 7.7 explizit beschrieben.

Definition(Konstruktion des Pushout-Komplements)
Nachdem die Gluing-Condition geprüft wurde und die Gleichung erfüllt wird, kann man das eindeutige Pushout-Komplement konstruieren. Der neue Graph D wird durch $D = (V_D, E_D, s_D, t_D)$ festgelegt.

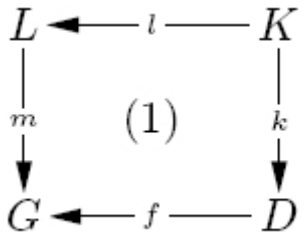

Abbildung 6: Pushout-Komplement

Explizit wird die Pushout-Komplement-Konstruktion definiert durch:

- $V_D = (V_G \backslash m_V(V_L)) \cup m_V(l_V(V_K))$;

- $E_D = (E_G \backslash m_E(E_L)) \cup m_E(l_E(E_K))$;
- $s_D = s_G|_{E_D}, t_D = t_G|_{E_D}$;
- $k_V(v) = m_V(l_V(v))$ für alle $v \in V_K . k_E(e) = m_E(l_E(e))$ für alle $e \in E_K$;
- f ist eine Inklusion;
- im Falle von getypten Graphen muss gelten, $type_D = type_G|_D$.

Definition(Graphtransformationssystem)
Ein Graphtransformationssystem $GTS = (P)$ besteht aus einer Menge von Graphproduktionen. Ein getyptes Graphtransformationssystem $GTS = (TG, P)$ besteht aus einem Typgraphen und einer Menge von getypten Graphproduktionen.

Definition(Graphgrammatik)
Eine (getypte) Graphgrammatik $GG = (GTS, S)$ besteht aus einem (getypten) Graphtransformationssystem GTS und einem (getypten) Startgraphen S.

Durch die Anwendung von Graphtransformationsregeln einer Graphgrammatik in beliebigen Regelanwendungssequenzen, beginnend mit einem Startgraphen, ist eine Menge von Graphen ableitbar. Jene Menge von Graphen bezeichnet man als visuelle Sprache: $L(GG) = \{G | S \Rightarrow^* G\}$

Verwendung kann die Theorie u.a. auch im Bereich sicherheitskritischer Anwendungen finden, da bei einem gegebenen Match ein deterministisches Verhalten durch Anwendung einer Graphtransformationsregel gewährleistet wird. Das Ergebnis der Regelanwendung ist eindeutig bis auf Isomorphie. Diverse Katastrophenszenarien zur Modellierung sicherheitskritischer Bereiche auf Basis der Graphtransformationstheorie wurden bereits am Lehrstuhl für „Theoeretische Informatik/Formale Spezifikation" an der TU Berlin spezifiziert und veröffentlicht.

5.8. Allgemeine Anwendungsbedingungen

In einem speziellen Kontext kann die Existenz oder Nichtexistenz von Knoten oder Kanten eines Graphen kontrolliert werden. Für jedes Auftreten eines Graphmusters „X" im Bereich des Matches muss zusätzlich überprüft werden, ob mindestens ein Graphmuster „C_i" im Bereich des Matches gefunden wurde. Bevor ich in den folgenden Abschnitten auf die beiden speziellen Fälle der negativen und positiven Anwendungsbedingungen eingehe, möchte ich kurz die Definition der allgemeinen Anwendungsbedingungen gemäß [$FAGT$] ansprechen.

Definition(Allgemeine Anwendungsbedingungen)
Eine atomare Anwendungsbedingung über einem (getypten) Graph L ist von der Form $P(x, \vee_{i \in I} x_i)$, wobei $x : L \to X$ und $x_i : X \to C_i$ mit $i \in I$ für eine Indexmenge I (getypte) attributierte Graphmorpshismen sind. Eine Anwendungsbedingung über L ist

ein boolescher Ausdruck über atomaren Anwendungsbedingungen über L.

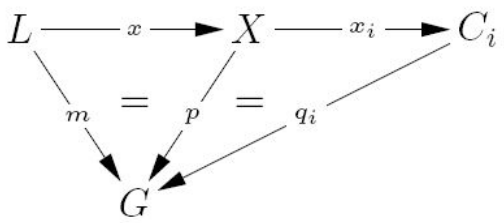

Abbildung 7: Allgemeine Anwendungsbediingung

Ein (getypter) Graphmorphismus $m : L \to G$ erfüllt eine Anwendungsbedingung acc, geschrieben $m \models acc$, wenn

- $acc = true$;
- $acc = P(x, \vee_{i \in I} x_i)$ und, für alle injektiven (getypten) Graphmorphismen $p : X \to G \in \mathcal{M}'$ mit $p \circ x = m$, existiert ein $i \in I$ und ein injektiver (getypter) Graphmorphismus $q_i : C_i \to G \in \mathcal{M}'$ mit $q_i \circ x_i = p$;
- $acc = \neg acc'$ und m erfüllt nicht acc';
- $acc = \wedge_{i \in I} acc_i$ und m erfüllt alle acc_i mit $i \in I$;
- $acc = \vee_{i \in I} acc_i$ und m erfüllt mindestens eine acc_i mit $i \in I$

5.9. Negative Anwendungsbedingungen

Eine Berücksichtigung einer negativen Anwendungsbedingung bei (getypten) attributierten Graphtransformationen überprüft, ob ein injektiver Morphismus zwischen dem Graphen der negativen Anwendungsbedingung und dem Graphen G existiert. Eine gängige Abkürzung für eine negative Anwendungsbedingung ist der Begriff NAC, den ich im Laufe meiner Diplomarbeit verwenden werde. Die Definition der negativen Anwendungsbedingung ist Bestandteil der Implementierung in 7.11.1.

Definition(Negative Anwendungsbedingungen)
Eine simple negative Anwendungsbedingung ist von der Form $NAC(x)$, wobei $x : L \to X$ ein getypter Graphmorphismus ist. Ein getypter Graphmorphismus $m : L \to G$ erfüllt $NAC(x)$, wenn kein injektiver, getypter Graphmorphismus $p : X \to G$ mit $p \circ x = m$ existiert.

Eine atomare negative Anwendungsbedingung ist von der Form $N(x, \wedge_{i \in I} x_i)$, wobei $x : L \to X$ und $x_i : X \to C_i$, mit $i \in I$ (getypte) Graphmorphismen sind. Ein (getypter) Graphmorphismus $m : L \to G$ erfüllt $N(x, \wedge_{i \in I} x_i)$, wenn für alle injektiven (getypten) Graphmorphismen $p : X \to G$ mit $p \circ x = m$ kein $i \in I$ und kein injektiver (getypter)

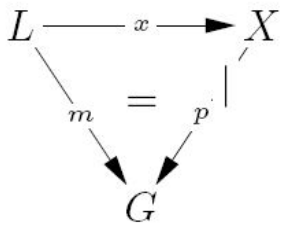

Abbildung 8: Efüllungsbedingung der NAC

Graphmorphismus $q_i : C_i \to G$ mit $q_i \circ x_i = p$ existiert.

Die x_i-Morphismen werden jedoch in unserem Falle nicht betrachtet und finden daher in unserer Implementierung keine Verwendung.

5.10. Positive Anwendungsbedingungen

Um die Existenz eines bestimmten Subgraphen in einem Graphen G zu gewährleisten, bzw. zu überprüfen, können ebenfalls positive Anwendungsbedingungen definiert werden, denn für jede NAC kann eine äquivalente PAC ermittelt werden. In [$FAGT$] heißt es:

Für jede atomare negative Anwendungsbedingung existiert eine äquivalente Anwendungsbedingung: $N(x, \wedge_{i \in I} x_i) \equiv \wedge_{i \in I} P(x_i \circ x, e)$, wobei e ein Ausdruck mit einer leeren Indexmenge ist. Analog kann ebenfalls zu $NAC(x) \equiv P(x, e)$ auch $PAC(x)$ für eine Anwendungsbedingung $\neg P(x, e)$, d.h. $PAC(x) = \neg P(x, e) = \neg NAC(x)$ definiert werden. Der Morphismus m erfüllt die PAC, wenn ein injektiver, getypter Graphmorphismus $p : X \to G$ mit $p \circ x = m$ existiert. Eine Beschreibung des Codes bezüglich der positiven Anwendungsbedingungen erfolgt in 7.11.2.

5.11. Parallele Unabhängigkeit

Bei der Frage der Auswirkungen der Ausführung einer (getypten) attributierten Graphtransformationsregel auf parallel ablaufende Regelprozesse berücksichtigt man eine Analyse einer parallelen Unabhängigkeit zweier Regeln. Bei der parallelen Unabhängigkeit darf die eine Regel keine Elemente löschen, die von der anderen Regel benötigt werden. Parallele Unabhängigkeit wird in der Implementierung aus Gründen der Effizienzsteigerung verwendet. Die Begriffe der parallelen Unabhängigkeit und der sequentiellen Unabhängigkeit führen zum Local Church-Rosser Theorem, auf welches ich nicht näher eingehen werde. Eine ausführliche Erklärung zur sequentiellen Unabhängigkeit (getypter) attributierter Graphtransformationsregeln und des Local Church-Rosser Theorems ist im Buch [$FAGT$] in Kapitel 3.3 zu finden.

Definition(Parallele Unabhängigkeit)
Zwei direkte (getypte) Graphtransformationen $G \overset{p1,m1}{\Rightarrow} H_1$ und $G \overset{p2,m2}{\Rightarrow} H_2$ sind parallel unabhängig, wenn alle Knoten und Kanten in der Schnittmenge der beiden Matches

Gluing-Punkte in Bezug auf beide Transformationen sind. Es muss gelten: $m_1(L_1) \cap m_2(L_2) \subseteq m_1(l_1(K_1)) \cap m_2(l_2(K_2))$

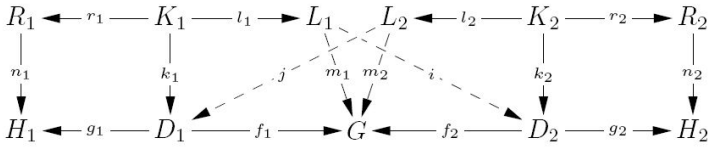

Abbildung 9: Parallele Unabhängigkeit zweier Graphtransformationsregeln

5.12. Constraints

Zur Formulierung bestimmter Eigenschaften eines Graphen besitzen sogenannte Constraints für Graphen eine Relevanz in der Hinsicht, daß man überprüfen kann, ob ein Graph G einen bestimmten Subgraphen G' enthält oder nicht. Hier erfolgt eine Aufteilung in Konklusion (C) und Prämisse (P). Die Umsetzung der Implementierung der Constraints ist in 7.11.3 beschrieben. Es folgt die Definition der Graph-Constraints gemäß [$FAGT$]:

Definition(Graph-Constraint)
Ein atomarer (getypter) Graph-Constraint ist von der Form $PC(a)$, wobei $a : P \to C$ ein (getypter) Graphmorphismus ist. Ein (getypter) Graph-Constraint ist eine boolesche Formel über atomare (getypte) Graph-Constraints. Das bedeutet, daß ein Graphconstraint „true" existiert und daß jeder atomare(getypte) Graph-Constraint ein Graph-Constraint ist, und für (getypte) Graph-Constraints c und c_i mit $i \in I$ für eine Indexmenge I, $\neg c$,$\wedge_{i \in Ic_i}$, und $\vee_{i \in Ic_i}$ sind (getypte) Graph-Constraints:

Ein (getypter) Graph G erfüllt einen (getypten) Graph-Constraint c, geschrieben $G \models c$, wenn

- $c = true$;
- $c = PC(a)$ und, für jeden injektiven (getypten) Graphmorphismus $p : P \to G \in \mathcal{M}'$ existiert ein injektiver (getypter) Graphmorphismus $q : C \to G$, sodaß $q \circ a = p$;
- $c = \neg c'$ und G erfüllt nicht c';
- $c = \wedge_{i \in Ic_i}$ und G erfüllt alle c_i mit $i \in I$;
- $c = \vee_{i \in Ic_i}$ und G erfüllt einige c_i mit $i \in I$.

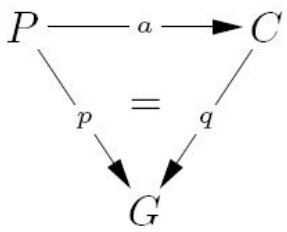

Abbildung 10: Erfüllungsbedingung des Constraints

Zwei (getypte) Graph-Constraints c und c' sind äquivalent, denotiert durch $c \equiv c'$, wenn für alle (getypten) Graphen G, $G \models c$ wenn und nur wenn $G \models c'$.

Die in diesem Kapitel formal fundierte Theorie der getypten attributierten Graphtransformation dient als Grundlage für eine Implementierung mit der Software Mathematica. Die Reihenfolge der theoretischen Konzepte wird bei der folgenden Erläuterung der Implementierung beibehalten. Illustriert werden die einzelnen Teilbausteine anhand einiger Beispiele in Form von Screenshots.

6. Mathematica

In dem folgenden Kapitel werde ich auf die proprietäre Software Mathematica näher eingehen, in den ersten Abschnitten allgemeine strukturelle Eigenschaften und besondere Vorteile der in Wolfram Mathematica integrierten Programmiersprache im Vergleich zu anderen Programmiersprachen erläutern. Es folgt in Abschnitt 6.1 ein Hinweis auf die beiden Schnittstellen MathLink und J/Link, die eine Konvertierung in C++, bzw. Java bereitstellen und so eine Kommunikation der Mathematica-Implementierung der getypten attributierten Graphtransformation mit Projekten im Bereich der visuellen Sprachen gewährleisten können. Anschließend wird die Funktionalität der Parallelisierung durch Mathematica erläutert. Die Vorteile und Optionen der graphischen Benutzeroberfläche der Software Mathematica werden in Abschnitt 6.3 kurz angeführt, bevor in Abschnitt 6.4 auf die Eclipse-Umgebung der Wolfram Workbench eingegangen wird. Die Notationen für die Zeitmessung in Mathematica sind vor allem für die in Kapitel 8 beschriebenen Benchmarks relevant. In Kapitel 7 werden die Implementierungen der Graphtransformationsthoerie beschrieben. Anhand zweier konkreter Benchmark-Beispiele werde ich im nächsten Kapitel 8 die Ergebnisse der Programmierung verdeutlichen.

Die plattformunabhängige und kommerzielle Software Mathematica des Unternehmens Wolfram Research wurde erstmalig im Jahre 1988 auf dem Markt eingeführt. Die Entwicklung der Software begann bereits im Jahre 1986. Das Softwareprodukt trägt den Namen des Firmengründers Stephen Wolfram. Der Erfolg der Software Mathematica wird auch angesichts weltweiten Verbreitung ersichtlich. So ist das Produkt Mathematica mittlerweile ein Standard in allen Vertretungen der US-Regierung und Bestandteil der Lehre in den 50 bedeutendsten Universitäten der Welt. Im Vergleich zur Software Matlab [*MATL*] bietet die Software Mathematica eine bessere Unterstützung von kontinuierlichen Funktionen.

Mathematica vereint u.a. die Vorteile der prozeduralen, objektorientierten, funktionalen und regelbasierten Programmiersprachen und bietet für den Nutzer zahlreiche Tools auch für die Anbindung anderer Programmiersprachen wie Java, C++ an. Zu den Bestandteilen der Software Mathematica gehören eine GUI (siehe 6.3), ein Computer-Algebra-System zur symbolischen Verarbeitung von Gleichungen, eine Numerik-Software für die Bearbeitung von mathematischen Gleichungen. Mathematica weist zahlreiche Eigenschaften der zuvor erwähnten verschiedenen Programmiersprachen auf. Zur Zeit wird die Version 7.0.1 weltweit lanciert. Kostenfrei ist lediglich der Mathematica Player 7.0, der jedoch für das Projekt im Zuge dieser Arbeit mit Mathematica durch seine Einschränkung bezüglich der Funktionalitäten nicht ausreichend gewesen ist.

Das sogenannte Front End als graphische Benutzeroberfläche und der Mathematica Kern (Kernel) bilden die grundlegende Struktur von Mathematica. Der Kernel bearbeitet Anfragen des Front End und gibt die errechneten Ergebnisse an die Benutzeroberfläche zurück, wobei dem Benutzer individuell überlassen ist, ob er ein sogenanntes „Notebook Interface" als Teil der graphischen Benutzeroberfläche oder ein text-basiertes Interface als Eingabebereich verwendet. Die sogenannten Notebook-Dateien agieren als interaktive

Dokumente, die Text, Tabellen, Formelsatz, Graphik, Töne, Berechnungen, Simulationen und Animationen in einem Benutzerinterface kombinieren und sind automatisch, hierarchisch in Abschnitte gegliedert. Der Benutzer kann Gruppen dieser Abschnitte in den Notebook-Dateien durch einen einfachen Mausklick ausblenden. Ein Interpreter sorgt für die Auswertung des Programmcodes, wobei die Ergebnisse und eventuellen Programmierfehler angezeigt werden, und ermöglicht dadurch eine interaktive Programmierung. MathLink fungiert als Kommunikationsschnittstelle, ermöglicht die Verbindung zwischen dem Mathematica Kernel und der Benutzeroberfläche [$MATH$].

Die Mathematica-Software stellt zudem eine benutzerfreundliche Kommunikation mit Excel-Projekten zur Verfügung.

Ein Vorteil der Nutzung von Mathematica ist die Möglichkeit der direkten Einbindung von Sonderzeichen (siehe 11), die u.a. auch über eine Menüleiste in der GUI ausgewählt werden können. Eine Auswahl verschiedener Paletten erhält man über den Menüpunkt „Palettes" in der oberen Leiste.

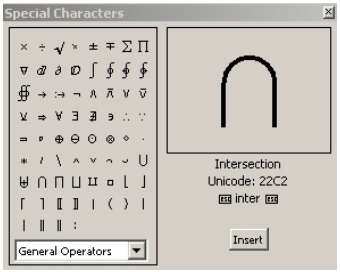

Abbildung 11: Sonderzeichen

Eine bibliotheksähnliche Struktur zur Kapselung von ausführbarem Code kann durch sogenannte Mathematica Packages realisiert werden. Es erfolgt eine Trennung zwischen der Menge an Funktionen, die in einem Package definiert und gespeichert werden, und der eigentlichen Auswertung, die in den Notebook-Dateien erfolgt. Ein Package dient demnach lediglich als Input und muss in die entsprechenden Notebook-Dateien integriert werden. Eine derartige Integration wird u.a. durch die Notation

```
Needs["Combinatorica`"];
Needs["Morphism`"];
```

umgesetzt. Benutzerunterstützende Tools sind u.a. ein in die GUI integriertes Tutorial, sowie ein Debugger.

6.1. MathLink und J/Link

MathLink ist eine offene API in C und bietet dem Benutzer die Möglichkeit individuell eine Art Front End für Mathematica zu entwickeln, um auf externe Programme zugreifen zu können. Wolfram Research bietet u.a. mit J/Link eine Integration von Java in

Mathematica und umgekehrt an, so daß Daten konvertiert und entsprechend übermittelt werden können. Die Funktionalitäten des J/Link-Interfaces können in einem späteren Stadium für eine Anbindung der Mathematica-Implementierung an eine GUI in Eclipse in Betracht kommen. Stub-Functions sind in C geschrieben und gewährleisten die Kommunikation mit dem MathLink-Interface. Die Abbildungen 12 und 13 illustrieren die Kommunikation des Kernel mit den Schnittstellen MathLink und J/Link sowie mit einer Notebook-Datei.

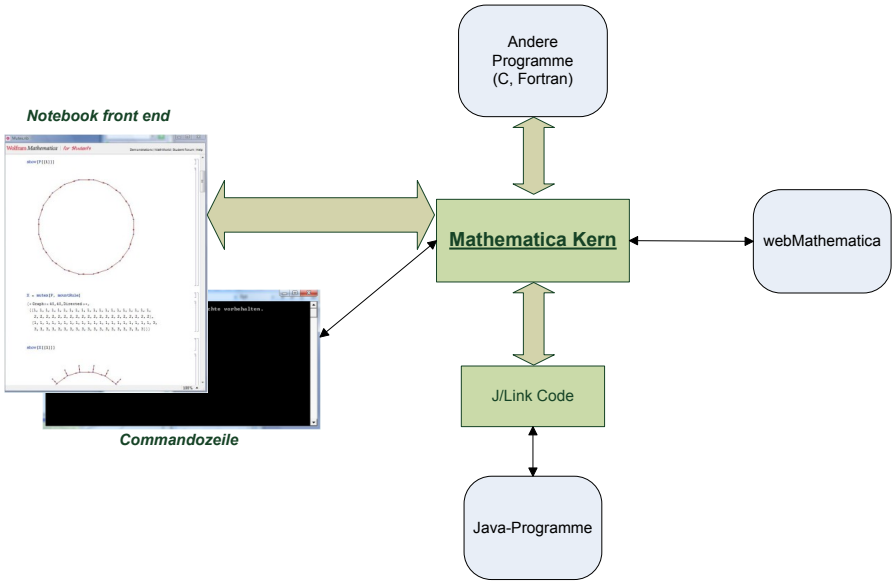

Abbildung 12: Mathematica Struktur

Außerdem werden sie als „native methods" in einer Klasse NativeLink definiert. Die Klasse KernelLink bezieht sich auf ein higher-level Interface. Grundsätzlich kann man konstatieren, daß das Toolkit J/Link eine transparente Benutzung von Java-Code in Mathematica und umgekehrt ermöglicht. J/Link enthält drei Pakete: com.wolfram.jlink , com.wolfram.jlink.ui und com.wolfram.jlink.util. Das erste Package enthält die Klassen und Schnittstellen, auf die der Anwender am häufigsten zugreifen muss. Im zweiten Package befinden sich einige Klassen, die in Programmen mit nicht-trivialen Benutzerschnittstellen von Interesse sind. Im letztgenannten Paket sind spezielle Klassen wie LinkSnooper enthalten. LinkSnooper zeigt den Datenverkehr zwischen zwei Programmen an.

Dem Benutzer ist es überlassen, ob er sich für die higher-level-Ebene (Klasse KernelLink) oder lower-level-Ebene (Klasse MathLink) entscheidet. Die Schnittstelle KernelLink erweitert hierbei die Schnittstelle MathLink und stellt weitere Funktionen für den Benutzer zur Verfügung, die lediglich relevant sind, wenn die korrespondierende Gegenseite ein Mathematica Kernel ist. MathLink gilt als wichtigere Schnittstelle, enthält durch die Vererbung selbstverständlich aber nicht alle im Interface KernelLink realisierten Operationen.

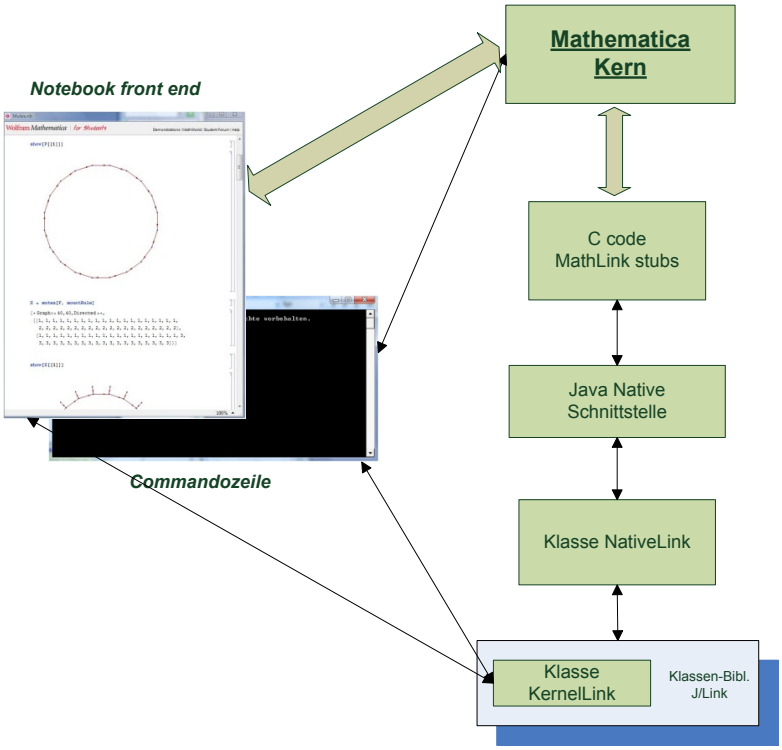

Abbildung 13: Interface J/Link

Um ein Linkobjekt zu definieren, wird die Klasse MathLinkFactory benötigt, welche die statischen Methoden createMathLink(), createKernelLink() und createLoopbackLink() enthält. Ein Beispielcode für die Definition eines KernelLink-Objektes könnte wie folgt aussehen:

KernelLink ml = MathLinkFactory.createKernelLink‘(”-linkmode launch -linkname ’c:\\program files\\wolfram research\\mathematica\\6.0\\mathkernel”’)

MathLink kann auch zum Aufrufen individueller Funktionen in einem externen Programm verwendet werden. Über MathLink kann der Quellcode generiert und in dem eigenen Programm integriert werden. Mit der Notation

```
Install["command"]
```

werden ein externes Programm aktiviert und Mathematica-Definitionen installiert. Die Funktionen, die in dem externen Programm integriert sind, können durch diese Notation in Mathematica aufgerufen werden. Der von Mathematica angebotene Befehl

```
Uninstall[link]
```

läßt das externe Programm terminieren und deinstalliert Definitionen, die sich auf Funktionen im externen Programm beziehen.

6.2. Parallelisierung

Die Einführung der Mathematica-Version 7 hat die Möglichkeit der Nutzung des Programmes für „parallel computing" optimiert, wobei zahlreiche neue Funktionen und Algorithmen für Parallelisierungen dem Nutzer zur Verfügung stehen. Automatische Parallelisierung, Parallelisierung von Datenstrukturen, Parallelisierung für Shared Memory und die Synchronisation sind Teil der parallelen Funktionalität. Im Vergleich zur Vorgängerversion 6 wird dem Nutzer der neuen Software die vierfache Rechenleistung durch 4 Compute-Kerne und die integrierte Parallelisierung angeboten. Eine hohe Kompatibilität ermöglicht eine Bearbeitung unter Unix-, Linux-, Windows- und Macintosh-Plattformen und gewährleistet eine Arbeit mit Multikern-Rechnern, mit Grid-Systemen oder in einem heterogenen Netzwerk.

6.3. GUI

Die Benutzeroberfläche von Mathematica bietet dem Benutzer zahlreiche Funktionalitäten für eine benutzerfreundliche und übersichtliche Verwendung. In der oberen Leiste der GUI stehen dem Benutzer 10 Rubriken zur Verfügung. Die Rubrik „File" dient der administrativen Aufgabe, der Verwaltung der Mathematica-Dateien und bietet dem Benutzer eine Übersicht der zuletzt bearbeiteten Dateien an.

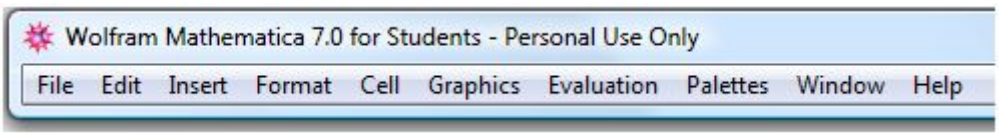

Abbildung 14: GUI

Das Ausführen der Dateien erfolgt über das Menü „Evaluation", wobei hier die Möglichkeit besteht, eine komplette Notebook-Datei auszuwählen, oder einzelne Fragmente, durch eckige Klammern abgegrenzte Bereiche, auszuführen. Auch per Short-cut „shift + enter" können Fragmente gezielt kompiliert werden. Desweiteren befinden sich hier Konfigurationsoptionen für den Kernel von Mathematica, sowie die Debugging-Funktion zum Testen der Implementierung. In Abschnitt 6.4 wird erläutert, aus welchen Gründen für das Testen und für die Verwaltung der Projektdateien das Eclipse-Plugin der Wolfram Workbench ausgewählt wurde.

Sehr ausführlich und mit zahlreichen Beispielen versehen bietet die Dokumentation unter der Rubrik „Help" dem Benutzer eine optimale Stütze für die Programmierung mit

Mathematica. Die Visualisierung geht so weit, daß Mathematica auch ein sogenanntes „Drawing-Tool" anbietet, mit dem Graphen und andere Symbole gezeichnet werden können. Diese Funktionalität befindet sich unter der Rubrik „Graphics". Zu beanstanden ist leider die Benutzerunfreundlichkeit.

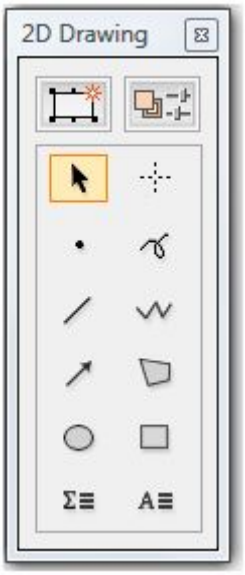

Abbildung 15: Drawing-Tool

6.4. Mathematica-Workbench für Eclipse

Um eine Anbindung von Javaprojekten an die Software Mathematica zu optimieren, habe ich mich für die Wolfram Workbench [*WW*] entschieden. Die Wolfram Workbench ist ein an die Mathematica-Lizenzen gebundenes Framework für Eclipse, einer integrierten Entwicklungsumgebung (IDE) für Wolfram Produkte wie Mathematica, gridMathematica und webMathematica.

Vorteile liegen in der Vereinigung verschiedener Aspekte. Einerseits fungiert die Wolfram Workbench als Editor, andererseits können auf der Eclipse-Ebene Fehleranalysen durch die Debuggingfunktion vereinfacht werden, wobei hier auf der Quellcode-Ebene getestet wird.

Man kann mit dieser integrierten Arbeitsgruppenumgebung unter Eclipse Java-Projekte und Mathematica-Projekte umsetzen und mit Hilfe der Versionsverwaltung (Concurrent Versions System, CVS) Teamarbeiten optimal gestalten. Unter der Rubrik „File" können mit „New" sogenannte „Source-", „Notebook-", „Scrapbook-" und „Package-Dateien" erstellt werden. Für die fehlerfreie Nutzung der Workbench müssen unter „Preferences" die Pfade des Mathemtica-Kernels korrekt gesetzt sein.

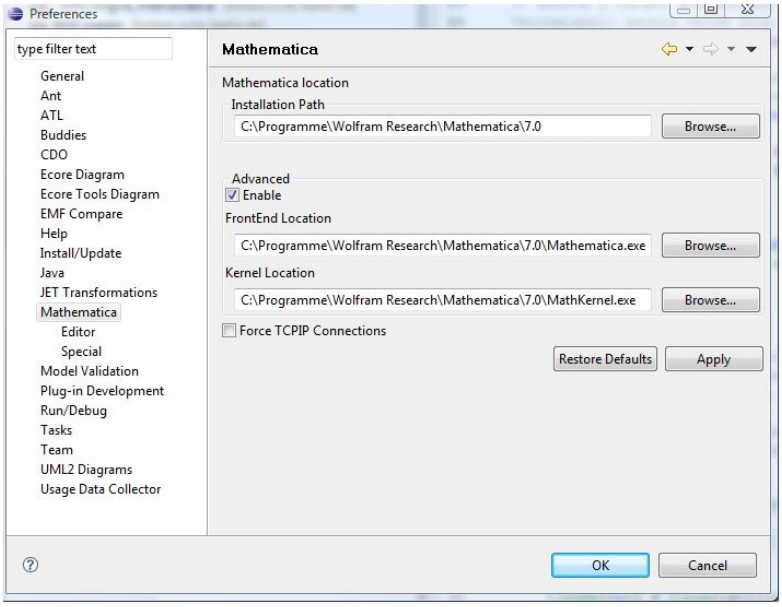

Abbildung 16: Konfiguration der Workbench unter Eclipse

Die Workbench fungiert unterstützend für die Konvertierung von Java-Code in Mathematica und umgekehrt, da Java-Sourcecode, Klassen für das Interface J/Link, DatabaseLink und GUI-Komponenten unterstützt werden. Eine visuelle Optimierung anhand einer Syntaxhervorhebung, des Hervorhebens lokaler Variablen, der Ausgabe von Fehlern in einem eigenen Fenster liefern weitere Argumente für die Benutzung dieses Produktes für die Mathematicaimplementierung. Unterstützt wird zusätzlich die unter Eclipse übliche Quellcodevervollständigung mittels der Tastenkombination „Strg + Leertaste". Über den Menüpunkt „Run" in der obigen Eclipse-Leiste kann die entsprechende Notebook-Datei geöffnet und gestartet werden. Auch wenn alle in dem Verzeichnis befindlichen Package-Dateien mitgeladen werden, müssen die jeweils benötigten Importe korrekt annotiert worden sein.

Die Mathematica-Workbench ist für „Premier Service"-Benutzer kostenfrei. Aktuell ist die Version 1.1 erhältlich, wird jedoch ähnlich wie bei dem Produkt Mathematica in regelmäßigen Abständen optimiert.

6.5. Zeitmessung unter Mathematica

Die Software Mathematica bietet für die Messung von Zeit verschiedene Funktionalitäten an. Für das Kapitel 8 der Effizienzanalyse der Implementierung der (getypten) attributierten Graphtransformation sind diese Funktionen von großer Bedeutung. Die Methode „Timing" evaluiert einen Ausdruck und liefert eine Liste der gemessenen Zeit in Sekunden

und ein Ergebnis der entsprechenden zu testenden Funktion zurück. Dieser Zeitwert beinhaltet jedoch nur die CPU-Zeit, die für den Mathematica-Kernel in Verwendung ist. Es erfolgt keine Berücksichtigung externer Prozesse, die bspw. über die Schnittstelle MathLink verbunden sind oder die durch das Mathematica-Front-end beansprucht werden. Desweiteren beeinflussen Formatierungen oder die Ausgabe des Ergebnisses nicht den Zeitwert. Die Problematik unsteter Zeitermittlungen durch eine Sequenz von Timing-Aufrufen kann durch die Methode „ClearSystemCache" gelöst werden.

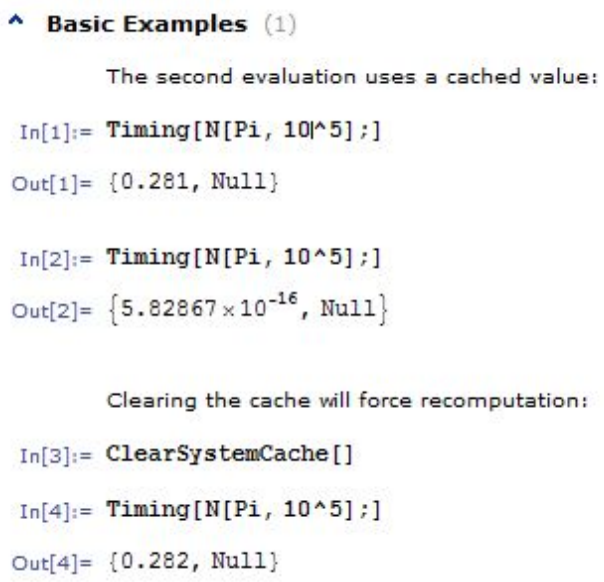

Abbildung 17: Beispiel für Methode ClearSystemCache

Diese Differenzen können aufgrund gespeicherter Werte durch interne Systemcaches entstehen.

6.5.1. Zeitmessung ohne graphische Benutzeroberfläche

Eine weitere Zeitersparnis ist durch ein Verzicht auf die graphische Benutzeroberfläche von Mathematica möglich. Eine Ausführung der Funktionen einer Notebook-Datei können entweder in der vom Betriebssystem bereitgestellten Shell oder durch Aktivierung des Kernel realisiert werden. Der Kernel läßt sich durch eine Ausführung der „MathKernel.exe"-Datei starten. Diese ausführbare Datei befindet sich im gleichen Verzeichnis wie die Datei „Mathematica.exe". Anschließend kann durch die von Mathematica bereitgestellte Funktion „SetDirectory" das entsprechende Verzeichnis ausgewählt werden, in dem sich die Projektdateien befinden. Der Code der Notebook-Datei muss nun in der GUI als „Initialization Cells" markiert und anschließend in eine Package-Datei konvertiert werden. Unter diesen Voraussetzungen kann eine Funktion ohne GUI in Ma-

thematica durch die Notation

```
Get["file"]
```

berechnet werden.

7. Implementierung der getypten attributierten Graphtransformation in Mathematica

Die zuvor beschriebenen theoretischen Grundlagen der getypten attributierten Graphtransformation in Kapitel 5 werden im folgenden Abschnitt eine Vorlage für die Implementierungen mit Mathematica sein.

Entsprechend der Strukturierung, bzw. Reihenfolge der Thematisierung der einzelnen theoretischen Fundamente wird in diesem Abschnitt nacheinander der wesentliche Teil der Implementierung angesprochen und anhand von zahlreichen Beispielen illustriert, welche Vorteile hieraus für den Benutzer dieser Software resultieren.

7.1. Struktur der Mathematica-Projektdateien

Das Projekt der unter Mathematica implementierten getypten attributierten Graphtransformation besteht aus den ausführbaren Notebook-Dateien und den Package-Dateien, in denen die relevanten Funktionen gespeichert sind. Für eine Regelanwendung entstehen Abhängigkeiten zwischen den einzelnen Dateien. Diese Verknüpfungen werden in der Abbildung 18 illustriert.

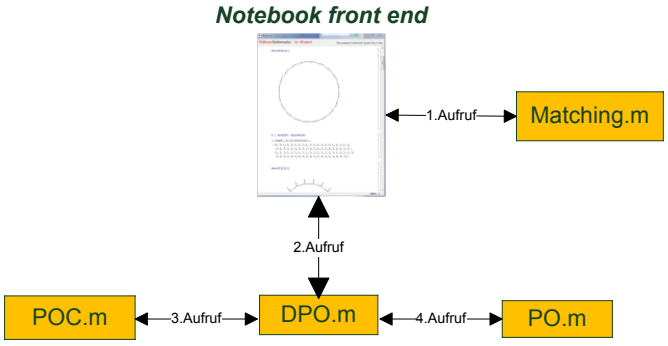

Abbildung 18: Dateienstruktur bei einer Regelanwendung

Um eine getypte attributierte Regel unter Mathematica anzuwenden, muss zuerst ein gültiger Match zwischen der linken Regelseite und dem Startgraphen gefunden werden. Die dafür benötigten Funktionen befinden sich in der Datei „Matching.m". Die eigentliche Regelanwendung erfolgt anschließend durch einen Aufruf der Datei „DPO.m". Es folgen Aufrufe der Dateien „POC.m" und „PO.m" innerhalb der Datei „DPO.m". Das Ergbnis der Regelanwendung oder etwaige Fehlermeldungen bei Konflikten bezüglich der Gluing-Condition werden in der Notebook-Datei ausgegeben.

Es gibt zwei Möglichkeiten der Deklaration von lokalen Variablen. Entweder man gibt durch `Module[{x,y,...},expr]` den Gültigkeitsbereich der lokalen Variablen an, oder man deklariert sie mit Hilfe von Funktionen und dazugehörigen Parametern.

```
Function[body]
```

Es existiert jedoch zusätzlich die Möglichkeit, lokalen Variablen bei der Konstruktion der Module initiale Werte der Variablen anzugeben:

```
Module[{x=0,...},expr]
```

7.2. Graphstrukturen

Für die Generierung und Visualisierung von Graphstrukturen sind die beiden von Mathematica angeboteten Funktionen „GraphPlot" und „GraphPlot3D" von Bedeutung, wobei hier lediglich auf die zweidimensionale Visualisierung zurück gegriffen wird. Diese Methode kann in den drei folgenden Varianten realisiert werden:

```
GraphPlot[{Subscript[v, i1]->Subscript[v, j1],
Subscript[v, i2]->Subscript[v, j2],...}]
GraphPlot[{{Subscript[v, i1]->Subscript[v, j1],Subscript[lbl, 1]},...}]
GraphPlot[m]
```

Der Parameter m steht in der dritten Funktionsvariante für eine adjacente Matrix. Während in der ersten Funktionsart Start- und Zielknoten angegeben werden, existiert in der zweiten Variante zusätzlich eine Assozierung mit sogenannten Labels für die Kanten. Die Funktion „GraphPlot" ist Teil der Visualisierungsfunktion „show" in der Package-Datei „Morphism.m". Die Funktionalität ist in unserem Fall auf die Operationen „Vertex-Labeling", „DirectEdges", „MultiedgeStyle" und „SelfLoopStyle" beschränkt.

```
GraphPlot[
```

```
G2, {VertexLabeling -> vLabeling, EdgeLabeling -> True,
 EdgeRenderingFunction -> Automatic, DirectedEdges -> True,
 MultiedgeStyle -> All, SelfLoopStyle -> All}]
```

option name	*default value*	
DirectedEdges	True	whether to show edges as directed arrows
EdgeLabeling	True	whether to include labels given for edges
EdgeRenderingFunction	Automatic	function to give explicit graphics for edges
Method	Automatic	the method used to lay out the graph
MultiedgeStyle	Automatic	how to draw multiple edges between vertices
PlotRangePadding	Automatic	how much padding to put around the plot
PackingMethod	Automatic	method to use for packing components
PlotStyle	Automatic	style in which objects are drawn
SelfLoopStyle	Automatic	how to draw edges linking a vertex to itself
VertexCoordinateRules	Automatic	rules for explicit vertex coordinates
VertexLabeling	Automatic	whether to show vertex names as labels
VertexRenderingFunction	Automatic	function to give explicit graphics for vertices

Abbildung 19: Operationen für GraphPlot

Die Operation „DirectedEdges", die, wie in dem Code-Fragment ersichtlich wird, auch Teil der Implementierung der Graphtransformation basierend auf Mathematica ist, dient der Wahl zwischen gerichteten und ungerichteten Kanten. Die Abbildung 19 zeigt eine Übersicht der zur Verfügung gestellten Operationen. In unserem Falle sind es gerichtete Kanten, symbolisiert durch den Pfeil. Die Attributierungen der Graphen für Knoten und Kanten werden durch die Funktionen „VertexLabeling" und „EdgeLabeling" umgesetzt. Eine Anpassung der Visualisierung der Kanten gemäß der individuellen Präferenz ist mit Hilfe der Operation „EdgeRenderingFunction" möglich. In unserem Fall wird diese Operation auf den initialen Wert „Automatic" gesetzt. In den Abbildungen 20, 21 und 22 sind einige Beispiele zu sehen, die durch die Operation „EdgeRenderingFunction" ermöglicht werden.

Das Mathematica-Interface J/Link, bzw. MathLink soll in einem späteren Stadium unter der GUI von Eclipse eine Visualisierung der Graphstrukturen ermöglichen und eine Anbindung an bestehende und zukünftige auf den Eclipse-Plugins GEF und EMF basierende, visuelle Projekte ermöglichen. Daher ist die Limitierung der visuellen Operationen in der Implementierung bewusst mit dem Hinblick auf die spätere Anbindung an Java-Projekte gewählt worden.

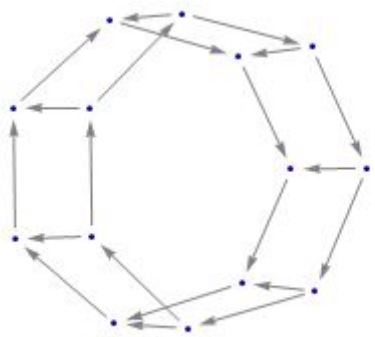

Abbildung 20: Variante 1 der EdgeRenderingFunction

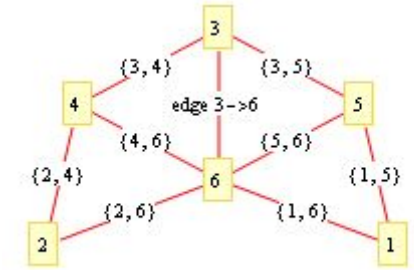

Abbildung 21: Variante 2 der EdgeRenderingFunction

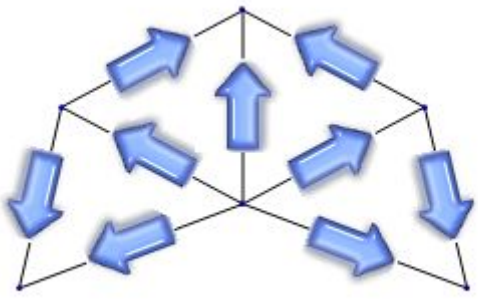

Abbildung 22: Variante 3 der EdgeRenderingFunction

Die Bibliothek „Combinatorica" erweitert die Funktionalität der Graphen von Mathematica durch über 450 weitere Funktionen in der Graphentheorie. Das Importieren dieser Bibliothek erfolgt entweder durch den Aufruf der Notation

```
<< Combinatorica`
```

oder durch Angabe zu Beginn einer Package-Datei mit der Notation.

```
BeginPackage["Morphism`",{"Combinatorica`"}]
```

Unter Eclipse ist es zudem möglich, diese Angaben direkt beim Erstellen einer neuen Package-Datei zu tätigen.

7.3. Typisierung und Attributierung

Ich erweitere nun Graphen zu (getypten) attributierten Graphen für die (getypte) attributierte Graphtransformationsimplementierung in Abschnitt 7.10. Die Realisierung der Attributierung von Graphen beschränkt sich auf die Angabe von sogenannten Labels für Knoten und Kanten. In Abschnitt 7.2 der Graphstrukturen wird bereits auf die Gewährleistung der Attributierung durch Labels der Funktion „GraphPlot" hingewiesen.

Typgraphen unterscheiden sich in der Code-Umsetzung von (getypten) attributierten Graphen nur in der Hinsicht, daß hier keine Definierung der Typisierung erfolgt. Der folgende Auszug aus der Datei „Sierpinski.m" zeigt einen Typgraphen mit einem Knoten v_1 und drei Kanten e_{12}, e_{23} und e_{31}.

```
typeGraph:=Module[{verticesTG,edgesTG,sourceTG,targetTG,TG},
verticesTG = {"v1"};
edgesTG = {"e12", "e23", "e31"};
sourceTG = {1, 1, 1};
targetTG = {1, 1, 1};
TG = objectGraph[verticesTG, edgesTG, sourceTG, targetTG]
]
```

Die Bedeutung der zentralen Funktion „objectGraph" wird im folgenden Abschnitt 7.4 über die Herangehensweise bei Morphismen mit Mathematica erläutert. Über einen solchen festgelegten Typgraphen werden die (getypten) attributierten Graphen über einen Morphismus verbunden. Eine nähere Beschreibung der Visualisierung eines solchen Morphismus erfolgt ebenfalls im nächsten Abschnitt.

7.4. Morphismen

Kernstück der Implementierung ist das Package „Morphism.m", welches die Funktion „objectGraph" enthält. Diese Funktion ist für die Konstruktion der Graphen in Mathematica zuständig. Eingabeparameter sind Knoten, Kanten, sowie target- und source-Informationen:

```
objectGraph = Function[{vertices, edges, source, target},...
```

Der Parameter „Vertices" enthält Objektknoten aller Typen, „Edges" sind die Kanten zwischen Knoten und werden durch ihre IDs angegeben, „Source" enthält die Listenelemente der Positionen der Source-Knoten, „Target" die Listenelemente der Positionen der Target-Knoten. Zwei Listen („Nodes" und „Edges") werden mit Hilfe der Parameter mit Werten gefüllt.

Eine effizientere, weil übersichtlichere und leichter verständliche Variante der Eingabe eines Graphen bietet die Funktion „aggGraph" an. Die Eingaben für die Elemente der Ziele und Quellen erfolgen nicht mehr durch die entsprechende Identifikationsnummer, sondern durch den Namen der Kante mit dem entsprechenden Kürzel für den Knoten. Für die Typisierung dieser optimierten Erstellung von Graphen wurde die Methode „calculateGraphMorphism" implementiert. Sie befindet sich ebenfalls in der Datei „Morphism.m".

Am Ende erfolgt die eigentliche Konstruktion des Graphen, wobei der Benutzer die Wahl zwischen gerichteten und ungerichteten Kanten hat.

```
G = Graph[Edges, Nodes, EdgeDirection -> True]
```

Ab einer Zahl von sieben Knoten werden die Knotenattribute nicht angezeigt. Mit Hilfe einer Tooltip-Implementierung lassen sich die genauen Angaben jedoch durch eine Mausbewegung auf den entsprechenden Knoten anzeigen.

In dem Package „Morphism.m" existieren zudem weitere Hilfsfunktionen und mehrere Funktionen zur Analyse in Bezug auf die Morphismen. Die Funktion „show" dient der Visualisierung der Graphen und ruft die von Mathematica vorgebene Funktion „GraphPlot" auf und bestimmt, daß ab einer Größe von 6 Knoten die Daten der Knoten durch ein Tooltip angezeigt werden.

```
show[G_Graph] :=
 Module[{G2, vLabeling},
  G2 = indexedGraph[G];
  If[V[G] <= 5, vLabeling = True, vLabeling = Automatic];
```

```
GraphPlot[
 G2, {VertexLabeling -> vLabeling, EdgeLabeling -> True,
  EdgeRenderingFunction -> Automatic, DirectedEdges -> True,
  MultiedgeStyle -> All, SelfLoopStyle -> All}]
]
```

Um eine Unterscheidung der Visualisierung zwischen einem Typgraphen und einem über einen Typgraphen getypten Graphen zu realisieren, wurde zusätzlich eine Funktion „showTG" implementiert. Das folgende Beispiel in Abbildung 23 zeigt einen Graphen, der über einen Typgraphen mit den drei Typen „ Lima, Kampala, Perth" getypt wird. Die Nummer zeigt die Position der Knoten im Typgraphen an. Die Visualisierung eines über einen Typgraphen getypten Graphen präsentiert einmal den getypten Graphen und einmal den Graphen mit den entsprechenden Typen. Die Übereinstimmung erfolgt über die Nummerierung.

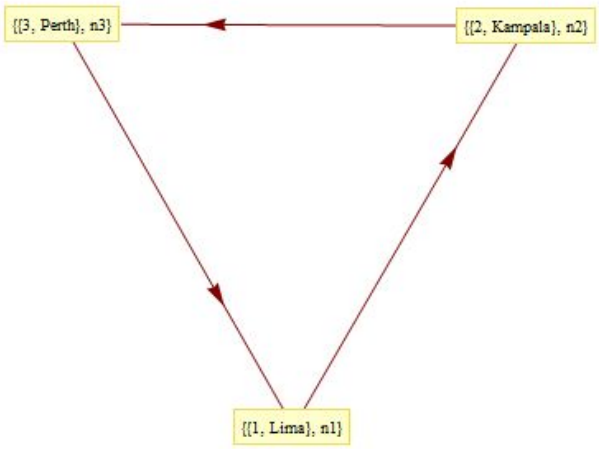

Abbildung 23: Funktion showTG/Getypter Graph

Die Funktion „showMorph" (siehe Abbildung 24) visualisiert die Morphismen und gibt beide Seiten der Morphismen als Graphik aus, wobei bei einem Morphismus f, der von G nach H geht, im Graphen H rot gekennzeichnet ist, von welchem Knoten gematcht wurde.

Die Funktion „indexedGraphUnion" wird benötigt, um bei einer späteren Vereinigung der beiden Graphen, dafür zu sorgen, daß die IDs für jeden Graphen getrennt betrachtet werden.

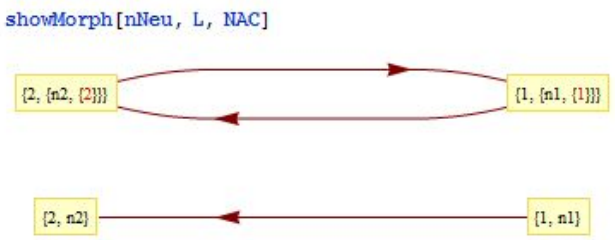

Abbildung 24: Funktion showMorph

Am Beispiel der Knoten möchte ich dies kurz anhand des Programmcodes illustrieren:

```
For[i = First[nodeIDs[G]], i <= n, i++,
    FirstNodes = Join[FirstNodes, {Nodes[[i]]}];
        ];
    FirstNodes = MapIndexed[insertIndex[First[#2], #1] &, FirstNodes];

For[i = n+1, i <= Last[nodeIDs[G]], i++,
SecondNodes = Join[SecondNodes, {Nodes[[i]]}];
        ];
    SecondNodes = MapIndexed[insertIndex[First[#2], #1] &, SecondNodes];
```

Die ersten Codezeilen der Funktion gewährleisten die Initialisierungen der lokalen Variablen. In den anschließenden For-Schleifen werden getrennt voneinander die Kanten und Knoten im Graphen G durchlaufen und jeweils das Listenfunktional „MapIndexed" angewandt. Nach den beiden Schleifenkonstruktionen erfolgt eine Konkatenation der beiden Listen „FirstNodes" und „SecondNodes" durch die Funktion Join aus der Mathematica-Bibliothek.

```
Nodes = Join[FirstNodes, SecondNodes];
```

Analog zu dieser Implementierung in Bezug auf die Knoten, muss auch bei den Kanten gewährleistet werden, daß die IDs der Kanten der beiden Graphen nicht addiert werden. Die Funktion „indexedGraphUnion" gibt nach Ausführung der Methode einen neuen Graphen H entsprechend der Struktur für Graphobjekte, die ich zuvor festgelegt hatte, aus.

7.5. Pushout-Konstruktion

Die Pushout-Konstruktion in unserer Mathematica-Implementierung orientiert sich an der in Kapitel 5.5 beschriebenen Definition und bezieht sich bereits auf die Charakterisierung für Graphen und ist Teil der Graphtransformation (siehe 5.7), weshalb die Notation der Definition der Konstruktion des Pushout-Objektes diejenigen Graphen und Morphismen enthält, die bei der Graphtransformation relevant sind. Die Formel $H = D \cup (R \backslash r(K))$ wird durch den ersten Abschnitt der Implementierung realisiert:

```
pushout[{K_Graph,typeK_List}, r_List, {R_Graph,typeR_List}, k_List,
{D_Graph,typeD_List}] :=

......

  {rVofVK, rEofEK} = r;
  {kVofVK, kEofEK} = k;
  VDobjects = nodes[D];
  EDobjects = edges[D];
  VR = nodeIDs[R];
  ER = edgeIDs[R];
  VRobjects = nodes[R];
  ERobjects = edges[R];
  (*H=D \[Union] (R\r(K)), disjoint union*)
  additionalVertexIDs = Complement[VR, rVofVK];
  additionalVertices = VRobjects[[additionalVertexIDs]];
  VHobjects = Join[VDobjects, additionalVertices];
  vertexOffSetD=V[D];
  nodeAmountR=V[R];

.......
```

Im ersten Schritt werden aus den Morphismen r und k jeweils die Elemente für Knoten und Kanten herausgefiltert und als Tupel

```
{rVofVK, rEofEK}
```

und

```
{kVofVK, kEofEK}
```

gespeichert. Die Bezeichnung „rVofVK" beschreibt die Knoten, die im Graphen der rechten Regelseite vorhanden sind und durch den injektiven Morphismus r erreicht werden. Diese Knoten werden durch die Regelanwendung nicht gelöscht.

Der Parameter „additionalVertexIDs" speichert die Elemente, die in „VR", aber nicht in „rVofVK" vorkommen. Anschliessend werden die Knoten mit Hilfe der Knoten-IDs gespeichert, die durch das Komplement berechnet wurden. Die Knoten des Graphen D werden disjunkt mit den in der lokalen Variablen „additionalVertices" gespeicherten Knoten vereinigt. Die disjunkte Vereinigung gewährleistet, daß die Elemente von $R\backslash K$ als neue Elemente hinzugefügt werden.

Die Berechnung der neuen Identifikationsnummern der Kanten für den Graphen H erfolgt durch Anwendung der Map-Funktion auf eine durch „Table" konstruierte neue Liste.

```
newEdgeIDsH =
   Map[(# + M[D]) &, Table[i, {i, Length[additionalEdgeIDs]}]];
```

Durch eine Konkatenation der bei den in den Variablen „kEofEK" und „newEdgeIDs" gespeicherten Listen kann der Morphismus n vervollständigt und gespeichert werden. Die Knotenwerte für den Morphismus n wurden bereits zuvor durch Anwendung der Hilfsfunktion „calculateNV" bestimmt.

```
nEofER = Join[kEofEK, newEdgeIDsH];
n = {nVofVR, nEofER};
```

Die Hilfsfunktion „calculateNewEdge" generiert eine aktualisierte Liste der Kanten eines neuen Graphen. Auf die Pushout-Konstruktion bezogen, werden die Kanten des Graphen h mit Hilfe des Morphismus n berechnet. Es folgt eine Angabe dieser Hilfsmethode, bevor ich mit der Funktion „pushout" fortfahren werde.

```
calculateNewEdge[e_List, nVofVR_List] :=
Module[{newEdge, nV},
 nV = (nVofVR[[#]]) &;
 newEdge = {{nV[src[e]], nV[tgt[e]]}, EdgeLabel -> label[e]}
 ]
```

In der Variablen „EHobjects" können die Kanten des Graphen H gespeichert werden. Dies erfolgt ebenfalls durch eine Konkatenation durch „Join".

```
EHobjects = Join[EDobjects, newEdgesH];
```

Eine Zuordnung zu der im Typgraphen definierten Typisierung wird durch die Codezeile

```
typeH=calculateTypeH[typeD,typeR,additionalVertexIDs,additionalEdgeIDs];
```

gewährleistet.

Der Morphismus f ist eine Inklusion und wird als Tupel angegeben. Das Pushout-Objekt in unserer Mathematica-Implementierung enthält einen Tupel, bestehend aus dem zuvor konstruierten Graphen H, der Typisierung für den Graphen H, sowie einem Morphismus f und einem Comatch n mit den beiden Komponenten „nVofVR" und „nEofVE".

```
(* f is inclusion D \[SubsetEqual] H *)
f = {nodeIDs[D], edgeIDs[D]};
PO = {{H,typeH}, n, f}
]
```

In der für die später folgenden Benchmarks essentiellen optimierten Variante der Pushout-Implementierung wird auf die Generierung des Morphismus g und des Co-Matches n verzichtet. Die letzte Zeile der Implementierung der Methode „pushoutO" verdeutlicht die Differenz zur standardmäßigen Codeumsetzung der Pushout-Konstruktion.

```
PO = {H,typeH} (* reduced *)
```

Ich möchte auf ein Beispiel der Pushout-Implementierung in der Datei „PO.nb" hinweisen. Abbildung 25 zeigt den Graphen A und den Morphismus zwischen dem Graphen A und dem Graphen B.

Es folgt die Visualisierung des Graphen B und des Morphismus zwischen den Graphen A und C in Abbildung 26.

Die Abbildung 27 zeigt den Graphen C mit zwei Knoten und zwei Kanten zwischen ihnen.

Die folgende Graphik 28 zeigt den Graphen des Pushout-Objektes. Zwischen den ersten beiden Knoten existieren zwei Kanten gemäß des Graphen C und eine Kanten zwischen dem zweiten Knoten und dem dritten Knoten gemäß der Struktur des Graphen B.

```
In[2]:= verticesA = {n1, n2};
        edgesA = {e12};
        sourceA = {1};
        targetA = {2};
        A = objectGraph[verticesA, edgesA, sourceA, targetA];
        typeA = {{1, 1}, {1}};
        show[A]
```


```
In[23]:= f = {{1, 2}, {1}}
```

Abbildung 25: Graph A und Morphismus f

```
In[24]:= verticesB = {n1, n2, n3};
         edgesB = {e12, e23};
         sourceB = {1, 2};
         targetB = {2, 3};
         B = objectGraph[verticesB, edgesB, sourceB, targetB];
         typeB = {{1, 1, 1}, {1, 1}};
         show[B]
```


```
In[31]:= g = {{1, 2}, {1}}
```

Abbildung 26: Graph B und Morphismus g

```
In[32]:= verticesC = {n1, n2};
         edgesC = {e12, e21};
         sourceC = {1, 2};
         targetC = {2, 1};
         Ce = objectGraph[verticesC, edgesC, sourceC, targetC];
         typeC = {{1, 1}, {1, 1}};
         show[Ce]
```

Abbildung 27: Graph C

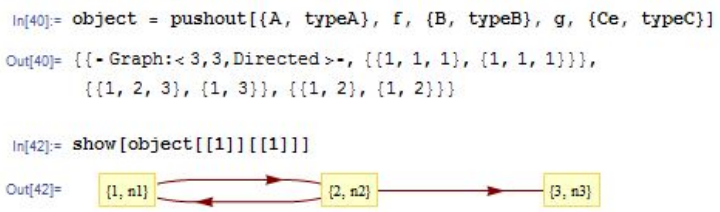

Abbildung 28: Pushout-Objekt

7.6. Pullback-Konstruktion

Die Pullback-Konstruktion gilt als duales Gegenstück zur Pushout-Konstruktion und ist Teil der Package-Datei „PB.m". Im Gegensatz zur Realisierung der Pushout-Konstruktionen in Mathematica werden Pullbacks für die Regelanwendungen unserer Benchmark-Beispiele nicht benötigt. Die Implementierung orientiert sich daher in Bezug auf die Namensgebung der Parameter an der Definition der Pullback-Theorie (siehe 5.6). Neben den drei Graphen B, C und D sind ebenfalls die beiden Morphismen f und g vor Anwendung der zentralen Funktion in der Datei „PB.m" „pullback" bekannt.

```
pullback[{B_Graph,typeB_List}, f_List, {C_Graph,typeC_List},
g_List, {D_Graph,typeD_List}]
```

Bestimmt werden muss der Graph A mit den beiden Morphismen k, ausgehend von A nach B, und dem Morphismus l von A nach C. Nach der Initialisierung der lokalen Variablen und der Schnittmenge der Knoten der beiden Morphismen f und g und der Kanten der beiden Morphismen f und g bestimme ich das Kreuzprodukt komponentenweise. Die Knoten und Kanten, die gemeinsam über die Elemente im Graphen D identifiziert werden, werden durch die Angabe aller möglichen Kombinationen im Pullbackobjekt auftauchen.

```
Do[
     flattenedPosNodesf = Flatten[Position[fVofVB, commonVertices[[i]]]];
     flattenedPosNodesg = Flatten[Position[gVofVC, commonVertices[[i]]]];
     AxBNodes = Join[AxBNodes,
     CartesianProduct[flattenedPosNodesf,flattenedPosNodesg]],
     {i,1, Length[commonVertices]}
  ];
```

Ausgehend vom Kreuzprodukt für die Knoten und Kanten, kann ich die beiden Morphismen k und l definieren.

```
Do[
  kv = Append[kv, AxBNodes[[i]][[1]]];
  lv = Append[lv, AxBNodes[[i]][[2]]],
  {i,1, Length[AxBNodes]}
  ];
  Do[
  ke = Append[ke, AxBEdges[[i]][[1]]];
  le = Append[le, AxBEdges[[i]][[2]]],
  {i,1, Length[AxBEdges]}
  ];
  k = {kv, ke};
  l = {lv, le};
```

Nun müssen noch die entsprechenden Knoten- und Kantenwerte für den Graphen A definiert werden. Berücksichtigt werden muss selbstverständlich die entsprechende Label-Belegung für Knoten und Kanten.

Am Ende der Implementierung erfolgt die eigentliche Konstruktion des Graphen A mit der dazugehörigen Typisierung.

```
A = Graph[AEdges, ANodes, EdgeDirection -> True];
typeA = calculateTypeA[typeB, k];
PB = {A,typeA}
```

Zu diesem Zwecke wird „calculateTypeA" mit den beiden Parametern der Typisierung für den Graphen B und des Morphismus k aufgerufen. Ich übernehme die Typisierung der korrespondierenden Knoten und Kanten im Graphen G.

Auszug aus der Funktion „calculateTypeA" für die Knoten des Graphen A:

```
Do[
typeAV[[i]] = typeB[[1]][[kV]][[i]],
{i,1,Length[typeAV]}
];
```

Die Berechnung der Typisierung der Knoten und Kanten des Pullbackobjektes A erfolgt komponentenweise, demzufolge die Schleifenkonstruktion für die Kanten den gleichen Aufbau besitzt. Am Ende der Methode erfolgt die Ausgabe der Typisierung.

```
typeA = {typeAV, typeAE}
```

Abbildung 29 gibt einen Einblick in die Berechnung und Visualisierung der Pullback-Implementierung mit Mathematica. Zur Erklärung möchte ich kurz auf die Klammerstruktur eingehen:

{KnotenID im Pullbackobjekt{{ Kombination durch kartesisches Produkt }, Label des korrespondierenden Knotens in Graph B }}.

Abbildung 29: Pullback-Beispiel

7.7. Doppelpushout-Konstruktion(DPO)

Für die Berechnung des Ergebnisgraphen der Doppelpushout-Konstruktion ist die Konstruktion des eindeutigen Pushout-Komplementes gemäß der Definition in 5.7 erforderlich. Ich habe bereits erwähnt, daß man das Pushout-Komplement nur konstruieren darf, wenn die Gluing-Condition erfüllt ist. Anhand der ersten Gleichung $V_D = (V_G \backslash m_V(V_L)) \cup m_V(l_V(V_K))$ möchte ich kurz demonstrieren, wie gering der Umfang des Mathematica-Codes für eine vollständige Umsetzung sein kann. $m_V(l_V(V_K))$ wird durch die Zeile

```
mVoflVofVK = Map[mV, lVofVK];
```

beschrieben, wobei „mV" eine Funktion ist, die punktweise auf die Liste „lVofVK" angewendet wird. Vervollständigt wird die Implementierung der zuvor angegebenen Definition durch das Code-Fragment

```
VD = Complement[VG, Complement[mVofVL,mVoflVofVK]] ;
```

Die Source- und Target-Verträglichkeit bzw. die Einschränkung von Quelle und Ziel im Graphen G auf die Kanten in D wird durch die Hilfsfunktionen „restrictToKeptEdges" und „restrictToKeptVertices" gewährleistet.

```
D1 = restrictToKeptEdges[G, ED];(*First delete Edges,
  since a node deletion may already delete edges*)
D = restrictToKeptVertices[D1, VD];
```

Löschende Graphtransformationsregeln erforden die Entfernung der entsprechenden Elemente unter Berücksichtigung der Gluing-Condition, auf die im folgenden Abschnitt eingegangen wird.

```
deletedNodeIDs=Complement[VG,VD];
  deletedEdgeIDs=Complement[EG,ED];
  nodeTypePositionsToDelete=Partition[deletedNodeIDs,1];
  typeDV=Delete[typeGV,nodeTypePositionsToDelete];
  edgeTypePositionsToDelete=Partition[deletedEdgeIDs,1];
  typeDE=Delete[typeGE,edgeTypePositionsToDelete];
```

Das Komplement bezüglich der Knoten- und Kantenmengen der beiden Graphen G und D berechnet die neue Liste der Knoten- und Kanten-IDs. Im nächsten Schritt ist eine Anpassung der Typisierung notwendig.

Ausgegeben wird das Pushout-Komplement:

```
POC = {{D,typeD}, {kV, kE}, {fV, fE}}
```

Implementierung der Gluing-Condition

In der Klasse „POC.m" befindet sich neben der Implementierung der Konstruktion des Pushout-Complements auch die Umsetzung der Gluing-Condition in Mathematica. Anhand der Gluing-Condition lässt sich sehr anschaulich erkennen, inwieweit die Vorteile einer funktionalen Programmiersprache genutzt werden können, bzw. wie die Definitionen der Gluing-Condition zum Teil direkt in Mathematica umgesetzt werden können. Diese funktionalen Eigenschaften führen zu einer offenkundig optimaleren Übersichtlichkeit, verringern enorm die Komplexität der Programmierung und erleichtern letztlich auch die Kommunikation bzw. Zusammenarbeit in einem Projekt.

Die Definition der Gluing-Points $GP = l_V(V_K) \cup l_E(E_K) = l(K)$ wurde in Mathematica wie folgt umgesetzt:

```
gluingPoints[l_List, K_Graph] :=
 Module[{gluingV, gluingE, gluingCond, nodesOfK, edgesOfK,
   gluingVOfK , gluingEOfK, gV, gE },
  {gluingV, gluingE} = l;
  gV = (gluingV[[#]]) &;
  gE = (gluingE[[#]]) &;
  nodesOfK = nodeIDs[K];
  edgesOfK = edges[K];
```

```
  gluingVOfK = Map[gV, nodesOfK];
  gluingEOfK = Map[gE, edgesOfK];
  gluingCond = {Union[Flatten[gluingVOfK]] ,
    Union[Flatten[gluingEOfK]]}
  ]
```

Beginnend mit den Bestimmungen der Werte für den Morphismus l für Knoten und Kanten, wird der Morphismus l mit Hilfe der Funktion Map komponentenweise auf die Knoten und Kanten des Graphen K ausgeführt.

Berücksichtigt werden muss zudem die Berechnung der sogenannten Identification-Points als Teilmenge der Gluing-Points. Der Code für die Definition der Identification-Points in Bezug auf die Knoten $IP = \{v \in V_L \,|\exists w \in V_L, w \neq v : m_V(v) = m_V(w)\}$ sieht wie folgt aus:

```
identificationPoints[m_List, G_Graph] :=
 Module[{mV, tallyV, tallyE, nodeAdaption , mE, actualVertice, actualEdge,
  i, posV, posE, s, edgeAdaption, identPos, x, y, oddE, oddV, flattenedE,
  flattenedV},
  {mV, mE} = m;
  oddV = {};
  oddE = {};
  (* list of recurrent elements *)
  tallyV = Tally[m[[1]]];
  tallyE = Tally[m[[2]]];
  (*for nodes:*)
  nodeAdaption = tallyV;
  For[i = 1, i <= Length[tallyV], i++,
   actualVertice = tallyV[[i]];
   If[(actualVertice[[2]] == 1),
    nodeAdaption = nodeAdaption \[Intersection] Delete[tallyV, i] ];
   ];......
```

Mit der Funktion Tally berechnet man die Anzahl der Vorkommen gleicher Werte, wobei in unserem Falle angegeben wird, wie oft die Knoten im Graphen G durch den Morphismus m erreicht werden. Anschließend werden alle Knoten aus der Hilfsvariablen tallyV gelöscht und die Schnittmenge mit der aktuellen Liste von tallyV gebildet, wobei durch die Schleife jedes Element der Liste tallyV erreicht und überprüft wird. Der Java-Code von AGG in Bezug auf die Identification-Points beträgt im Vergleich zur Mathematica-Implementierung 63 Zeilen. Die Funktion „isIdentificationConditionSatisfied()" befindet sich in der AGG-Klasse „Match.java".

```
......
flattenedV = Flatten[nodeAdaption, 2];
......
For[i = 1, i <= Length[flattenedV], i++,
   If[OddQ[i], oddV = Append[oddV,flattenedV[[i]]];
     ];
   ];
......
x = Map[posV, Flatten[oddV, 2]];
......
```

Die zweite For-Schleife in Bezug auf die Knoten ist für die Anpassung der Liste notwendig, denn die Elemente an allen geraden Positionen müssen aus der Liste entfernt werden. Ich erinnere daran, daß durch die Funktion Tally stets Tupel von Werten {KnotenID im Graphen G, Anzahl der Vorkommen durch den Match m} konstruiert wurden. Bei den Kanten verfährt man analog. Das Ergebnis ist ein Tupel, bestehend aus zwei Listen, eine für die Knoten und eine für die Kanten, die als Identification-Points bestimmt wurden.

```
identPos = {Union[Flatten[Map[posV, Flatten[helpV2, 2]], 2]],
    Union[Flatten[Map[posE, Flatten[helpE2, 2]], 2]]}
```

7.8. Matching

In der Package-Datei „Matching.m" werden die Matches zwischen zwei Graphen bestimmt, wobei man diese Funktionalität für die attributierte Graphtransformation aber auch für Matches zwischen NAC- bzw. PAC-Graphen und dem Graphen G verwenden kann. Traversiert wird in der Graphstruktur mit Hilfe der Breitensuche, auch Breadth First Search genannt. Inkludiert werden müssen die beiden Package-Dateien „Combinatorica" und „Morphism", ansonsten kann hier auf bestimmte, notwendige Funktionen nicht zugegriffen werden. Bestimmt werden alle möglichen Matches, die in einer in Mathematica üblichen Liste ausgegeben werden. Die Liste teilt sich auf in die Matches für die Knoten und für die Kanten. Ausschlaggebend für die Bestimmung der Morphismen ist nicht nur die Struktur des Graphen G, auf den die entsprechende Regel angewandt wird, sondern zusätzlich die Typisierung der einzelnen Knoten und Kanten. Die Übereinstimmung der Typen wird durch entsprechende Funktionen in der Implementierung gewährleistet.

In der Funktion „calculateMatches" wird vorab geprüft, ob die entsprechende Graphtransformationsregel Anwendungsbedingungen enthält, wobei bisher neben Constraints,

auch NACs und PACs in Mathematica umgesetzt wurden.

Die Funktion „matchesSimple" initialisiert den Morphismus m zwischen der linken Regelseite und dem Hostgraphen mit dem Wert 0.

```
mV = Table[0, {nodeAmountL}]; (*initialize mV with zeros*)
mE = Table[0, {edgeAmountL}]; (*initialize mE with zeros*)
```

Sollte der Hostgraph keine Knoten, respektive dann auch keine Kanten enthalten und der Graph L nicht leer sein, so existiert kein Morphismus. Ansonsten wird bei einer Abfrage in dieser Funktion überprüft, ob überhaupt Kanten in dem Hostgraphen vorhanden sind. Bei einer leeren Menge für Kanten wird direkt die Funktion „continueEdgeMatchesSimple" aufgerufen, ansonsten erfolgt der Funktionsaufruf der Methode „continueMatchesSimple", zu der ich als nächstes kommen werde.

Nach der Initialisierung des Morphismus m wird in der Methode „continueMatchesSimple" die erste Kante in der Liste der Kanten des Graphen L ausgewählt und bestimmt. Wichtig ist hier die Übereinstimmung der Typisierung, sodaß nur die Kanten in der lokalen Variablen „matchesEdges" herausgefiltert werden, die den gleichen Typen wie die erste ausgewählte Kante des Graphen L enthalten. Dies erfolgt durch Anwendung der von Mathematica bereitgestellten Methode „Position". Das Flattening auf das Ergebnis des Listenfunktionals „Position" korrigiert lediglich die Klammerstruktur der ausgegebenen Liste.

```
matchedEdges = Flatten[Position[typeGE, typeLE[[pos]]]];
```

Anschließend sucht die Funktion alle benachbarten Kanten mit Hilfe der Funktion „calculateAdjacentEdges" heraus. In jener Funktion werden zu Beginn alle IDs des Graphen G bestimmt, anschließend die Kanten mit Hilfe des Listenfunktionals „Select" herausgefiltert, deren Quell- und Zielverträglichkeit mit der aktuellen Kante erfüllt ist.

Ausschnitt aus der Hilfsfunktion „calculateAdjacentEdges":

```
adjacentEdges = Select[edgesG,(
(source[#,G]==source[edge,G])||
(source[#,G]==target[edge,G])||
(target[#,G]==target[edge,G])||
(target[#,G]==source[edge,G])
)&];
```

Die Schleifenkonstruktion dient der Überprüfung aller Kanten im Graphen G mit dem entsprechenden Typen. Die nachfolgenden Matches werden durch die Funktion „continueEdgesMatchesSimple" berechnet.

```
Do[
If[(source[pos,L] != target[pos,L])||(source[e,G]==target[e,G]),
mE[[pos]] = e;
mV[[source[pos,L]]] = source[e,G];
mV[[target[pos,L]]] = target[e,G];
adjacentEdges = calculateAdjacentEdges[pos,L];
(* delete current edge *)
remainingEdges2Match = Complement[adjacentEdges,{pos}];
newMatches = continueEdgeMatchesSimple[{L,typeL},{mV,mE},
remainingEdges2Match, {G, typeG}];
matches = Join[matches,newMatches]]
,{e,matchedEdges}
];
```

Das Verfahren der Breitensuche zum Durchlaufen der Knoten des Graphen reicht jedoch bei mehreren Zusammenhangskomponenten, d.h. bei Existenz mehrerer Teilgraphen, nicht aus. Daher wurde die Funktion „continueEdgeMatchesSimple" als eine der zentralen Funktionen u.a. für die Berücksichtigung von Teilgraphen umgesetzt. Der Funktion werden die Liste des Graphen L und die dazugehörige Typisierung als Parameter übergeben. Zusätzlich enthält die Methode eine Liste der Kanten der ersten Zusammenhangskomponente, im Zweifelsfall bei der Nichtexistenz mehrerer Teilgraphen nur die Liste der Kanten des einen Graphen sowie den Graphen G und die dazugehörige Typisierung. Beginnend mit der Intialisierung bspw. des lokalen Parameters „matches" folgt eine Abfrage, ob die Menge der noch zu matchenden Kanten der ersten Zusammenhangskomponente leer ist.

```
If[remainingEdges2Match=={}
```

Ist dies der Fall, so müssen isolierte Knoten und die Existenz weiterer Kanten in der gesamten Graphstruktur im Graphen L überprüft werden. Die Hilfsfunktion „calculateIsolatedNodeMatchesWithPriorities" befindet sich ebenfalls in diesem Paket und kalkuliert jene Knoten, die isoliert sind, ergo keine Kantenverbindungen vorzuweisen haben. Ein rekursiver Aufruf der Funktion „continueEdgeMatchesSimple" dient der Abarbeitung weiterer Kanten im Graphen L der linken Regelseite bzw. des Urbildes des Morphismus m. Anschließend wird der erste Eintrag der Liste „remainingEdges2Match" entnommen und der Rest in einer weiteren lokalen Variablen namens „remainingEdges2MatchNew" gespeichert, wobei Mathematica hier die Speicherung von Daten in mehreren lokalen Variablen in einer Zeile ermöglicht.

```
{edge,remainingEdges2MatchNew}=
   {First[remainingEdges2Match],Rest[remainingEdges2Match]};
```

Erneut erfolgt eine Abfrage. Ist der Wert der Kante im Morphismus m schon gesetzt, erfolgt ebenfalls ein rekursiver Aufruf der hier zu erläuternden Funktion. Der folgende Schritt ist eine Überprüfung, ob die Knoten, die als Ziel oder Quelle einer Kante identifiziert werden, in der Knotenliste des Morphismus m bereits gesetzt wurden. Hierfür werden vier verschiedene Kombinationen durch eine Switch-Konstruktion abgefragt, wobei der erste Fall eigentlich nicht eintreten wird, aber als Alternative für den Start-Fall die Funktionalität der Methode erhöhen soll. Für den Fall, daß nur der Startknoten gesetzt ist, werden mit Hilfe der Funktion „calculateTargetAdjacentEdges" die benachbarten Kanten bestimmt. Analog dazu erfolgt die gleiche Berechnung für den Fall, daß nur der Zielknoten bereits festgelegt wurde. Die benachbarten Kanten werden ab Zeile 205 auf die Kanten durch eine Select-Notation reduziert, die noch nicht gematched wurden. Um ein mehrmaliges Auftreten der Kanten-IDs zu verhindern, muss eine disjunkte Vereinigung der Menge der Kanten durchgeführt werden. Im letzten Teil der Funktion erfolgt die Belegung der letzten mit 0 belegten Stellen in den Listen der Kanten und Knoten des Morphismus m. Das Endresultat ist die Ausgabe aller möglichen Kombinationen an Matches.

Eine weitere Optimierungsfunktionalität ist eine priorisierte Herangehensweise an das Berechnen der Matches bei isolierten Knoten. Diese Optimierung befindet sich in der Funktion „calculateIsolatedNodeMatchesWithPriorities". Möchte man zwischen der standardmäßigen Funktion für isolierte Knoten und der optimierten Version wechseln, muss man in der Funktion „continueEdgeMatchesSimple" die Funktion auskommentieren, die man nicht wählen möchte. Benötigt wird die Hilfsfunktion „checkWeightOfNodes", die lediglich die Typisierung der Knoten und Kanten des Graphen L übergeben bekommt. Das minimale Vorkommen der Typisierung wird für den übergebenen Graphen ermittelt und am Ende ausgegeben. Man beschränkt sich aber bei der Verwendung lediglich auf die Funktion „checkWeightOfNodes", da eine Optimierung der Kanten anhand ihrer Typisierung bisher nicht realisiert wurde.

```
checkWeightOfEdges[typeX_List]:=
Module[{tallyE, tmpE, newTypeXE, minPosition, newTypeX, newX, i},
newTypeXE = typeX[[2]];
tallyE = Tally[newTypeXE];
tmpE = First[tallyE][[2]];
minPosition = First[tallyE][[1]];
For[i = 1, i <= Length[tallyE], i++,
        If[tmpE > tallyE[[i]][[2]],
           tmpE = tallyE[[i]][[2]];
            minPosition = tallyE[[i]][[1]]
  ];
```

```
   ];
   minPosition
]
```

Das Listenfunktional „Tally" wird auf die Liste der Typisierung angewandt. Es entsteht eine Liste von Tupeln mit einer Angabe der Typisierung und der Häufigkeit des Vorkommens. Über diese generierte Liste kann nun iteriert werden, um in der Schleifenkonstruktion die minimal auftretende Typisierung zu bestimmen.

Die Funktion „checkWeightOfEdges" ist jedoch implementiert und für spätere Erweiterungen eine Option. Eine weitere Optimierung auf Basis der Berechnung der Matches, indem lediglich ein einziger Match berechnet wird, führt bei der zeitlichen Messung der Berechnungen in Kapitel 8 zu einer deutlichen Verbesserung. Die Funktion „continueEdgeMatchesSimpleOneMatch" berechnet lediglich einen einzigen Match, der bspw. für die Abfrage der NACs ausreichend ist. Neben Anpassungen der rekursiven Funktionsaufrufe unterscheidet sich diese Funktion, einerseits durch zusätzliche Parameter und durch die folgenden Code-Zeilen von der bisher erläuterten Funktion „continueEdgeMatchesSimple".

```
If[newMatches != {},
match=newMatches[[1]];
If[CheckGC==True,
If[isGluingCondition[G, match, l, K, L]==True,
matches = Join[matches,newMatches];
Break[]
],
matches = Join[matches,newMatches];
Break[]
]
];
```

Diese zusätzlichen Abfragen gewährleisten, daß der Match auch die Gluing-Condition erfüllt. Sobald ein gültiger Match gefunden wurde, wird die um diesen Code befindliche Do-Schleife durch die „Break"-Notation verlassen. Der Wert der booleschen Variable bestimmt, ob diese Überprüfung durchgeführt wird. Es wurde bereits erwähnt, daß die Funktion „continueEdgeMatchesSimpleOneMatch" ebenfalls in der Datei „MatchConditions.m" Verwendung findet. In diesem Falle wird der Parameter „CheckGC" auf „false" gesetzt, so daß keine Überprüfung der Gluing Condition erfolgt.

7.9. Parallelisierung des Matching

Ich komme nun zu einer weiteren Optimierung der Regelanwendung, die Verwendung in der Sierpinski- und der Mutex-Implementierung findet. Voraussetzung hierbei ist die Nichtexistenz von Anwendungsbedingungen. Eine Sequenz von Regeln müßte bei der Existenz von Anwendungsbedingungen, seien es Constraints, positive Anwendungsbedingungen oder negative Anwendungsbedingungen, nach jeder Applikation erneut die Bedingungen überprüfen. Somit kommt eine direkte parallele Ausführung der Regeln nicht in Frage. Die Definition der parallelen Unabhängigkeit befindet sich in Kapitel 5.11.

Die Optimierung der Regelanwendung erfolgt durch das Verkleben der Regel zu einer parallelen Regel. Für jeden Match wird eine Regel als Kopie definiert. Zentrale Funktion in der Package-Datei „ParallelProduction.m" ist die gleichnamige Methode, der die komplette Regel und eine Liste der Matches als Parameter übergeben werden. Die parallel möglichen Matches werden komponentenweise einmal in der lokalen Variable „parallelMatchV" und die Kanten in der Variable „parallelMatchE" zwischengespeichert.

```
parallelMatchV=Flatten[Map[(#[[1]])&,matches]];
parallelMatchE=Flatten[Map[(#[[2]])&,matches]];
parallelMatch={parallelMatchV,parallelMatchE};
```

Abhängig von der Anzahl der Matches zwischen der linken Regelseite und dem Hostgraphen werden für alle Teile der Regel (L-Graph, K-Graph, R-Graph) die Typisierung mit der Hilfsfunktion „cloneTyping" geklont. Die Anzahl der möglichen Matches, repräsentiert durch den Parameter „matchesAmount", bestimmt hier die Anzahl der Listenelemente für die Typisierung.

```
parallelTypingL=cloneTyping[typeL,matchesAmount];
parallelTypingK=cloneTyping[typeK,matchesAmount];
parallelTypingR=cloneTyping[typeR,matchesAmount];
```

Anschließend werden ebenfalls die Knoten aller Teile der Regel gemäß der Anzahl der Matches geklont, bzw. durch „Table" neu aufgebaut. Diese Listen beinhalten die komplette Knotenstruktur mit den entsprechenden Labels. Das Flattening des Resultates korrigiert lediglich die Klammerstruktur. Die Funktion „verticesAndLabels" befindet sich in der Datei „Morphism.m" und generiert die Knotenelemente mit dem Label aus einem Graphen.

```
parallelNodesL=Flatten[Table[verticesAndLabels[L],{matchesAmount}],1];
parallelNodesK=Flatten[Table[verticesAndLabels[K],{matchesAmount}],1];
parallelNodesR=Flatten[Table[verticesAndLabels[R],{matchesAmount}],1];
```

Auch für die Kanten der Regelseite erfolgt ein getrenntes Klonen aller Graphen der Graphtransformationsregel. Die Veränderung der Kantenliste erfordert ein Offset zur Anpassung der Identifikationsnummern durch Einbeziehung der Hilfsmethode „positionsWithOffset".

```
parallelEdgesL=cloneEdges[edgesAndLabels[L],matchesAmount,singleOffSetL];
parallelEdgesK=cloneEdges[edgesAndLabels[K],matchesAmount,singleOffSetK];
parallelEdgesR=cloneEdges[edgesAndLabels[R],matchesAmount,singleOffSetR];
```

Die drei neuen Graphen werden nun mit den zuvor ermittelten Werten definiert, bzw. als Graphobjekte gespeichert und als Graphen mit gerichteten Kanten ausgewiesen.

```
parallelL=Graph[parallelEdgesL,parallelNodesL,EdgeDirection->True];
parallelK=Graph[parallelEdgesK,parallelNodesK,EdgeDirection->True];
parallelR=Graph[parallelEdgesR,parallelNodesR,EdgeDirection->True];
```

Nachdem bisher die Graphen der Regel kopiert und als neue Graphen abgespeichert wurden, fehlt noch die Bestimmung der injektiven Morphismen zwischen den Graphen K und L und zwischen den Graphen K und R. Maßgeblich für die Bestimmung der Morphismen ist auch in diesem Fall die Anzahl der Matches.

```
(* clone l and r *)
{lV,lE}=l;
{rV,rE}=r;
parallelMorphismL={cloneMorphismWithOffSet[lV,V[L] ,matchesAmount],
   cloneMorphismWithOffSet[lE,M[L] ,matchesAmount]};
parallelMorphismR={cloneMorphismWithOffSet[rV,V[R] ,matchesAmount],
   cloneMorphismWithOffSet[rE,M[R] ,matchesAmount]};
```

Die veränderte, geklonte Regel kann nun neu definiert werden und besteht aus fünf Listenelementen.

```
parallelRule={
{parallelL,parallelTypingL},parallelMorphismL,
{parallelK,parallelTypingK},parallelMorphismR,
{parallelR,parallelTypingR}};
```

Zuletzt erfolgen die Ausgabe der Regel und der dazugehörige parallele Match. Die Belegungen der Listenelemente „parallelRule" und „parallelMatch" sind bei Ausführung der Funktion „parallelProduction" in einer Notebook-Datei sichtbar.

```
{parallelRule,parallelMatch}
```

7.10. Getypte attributierte Graphtransformation

In diesem Teil des Kapitels möchte ich eine Übersicht über die beiden Varianten der Notationen für attributierte Graphtransformationsregeln geben. Die vorherigen Bereiche der Implementierungen legen dabei den Grundstein für die Definition von Regeln in Mathematica. Graphtransformationsregeln, Sequenzen von Graphtransformationsregeln können nun durch zusätzliche Angabe eines Startgraphen in Mathematica ausgeführt und visualisiert werden.

Als Beispiel einer Notation der nicht optimierten Variante wähle ich die Regel „take-Rule" der Graphgrammatik Mutex, auf die ich im Kapitel 8 der Effizienzanalyse explizit eingehen werde.

```
takeRule:=Module[{
L, R, K, typeL, typeK, typeR, l, r,
verticesR, edgesR, sourceR, targetR,
verticesL, edgesL, sourceL, targetL
},
verticesL = {"p", "r"};
edgesL = {"t", "req"};
sourceL = {2, 1};
targetL = {1, 2};
L = objectGraph[verticesL, edgesL, sourceL, targetL];
typeL = {{1, 2}, {3, 5}};

K = objectGraph[verticesL, {}, {}, {}];
typeK = {{1, 2}, {}};

verticesR = {"p", "r"};
edgesR = {"hb"};
sourceR = {2};
targetR = {1};
R = objectGraph[verticesR, edgesR, sourceR, targetR];
typeR = {{1, 2}, {2}};

l = {{1, 2}, {}};
r = {{1, 2}, {}};

{{L,typeL},l,{K,typeK},r,{R,typeR}, {{}, {{},{}}}}
]
```

Unterschiede liegen in der Angabe der Source- und Target-Informationen. In der optimierten Version werden anstatt IDs die entsprechende Kante, die gerade angesprochen wird, und der entsprechende Knoten als Label gespeichert. Die folgende Abbildung 30 ist ein Auszug aus der Notebook-Datei „UseApplyRule.nb" und zeigt einen Instanzgraphen als Beispiel der optimierten Variante der Regelnotation in Mathematica.

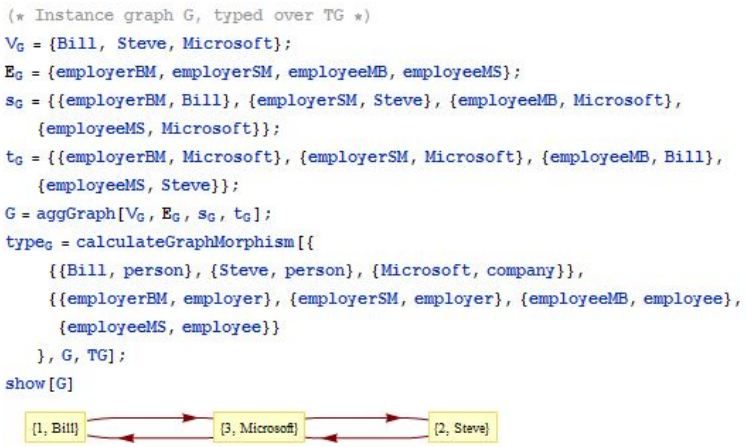

Abbildung 30: Beispiel der optimierten Regelangabe

7.11. Anwendungsbedingungen

Um Entscheidungen zu fällen, ob eine ausgewählte Regel auf einen Graphen angewandt werden soll, oder ein Graph eine bestimmte Struktur besitzen und demnach durch die entsprechende Regel nicht verändert werden soll, wurde das Konzept der Constraints und Anwendungsbedingungen definiert. Die Existenz oder Nichtexistenz bestimmter Knoten und Kanten kann durch dieses Konzept überprüft werden. Dieser Sachverhalt verdeutlicht die Mächtigkeit der Anwendung der Graphtransformationstheorie. In den folgenden beiden Unterkapiteln werde ich auf die Implementierung der negativen (siehe 7.11.1) und positiven Anwendungsbedingungen (siehe 7.11.2) sowie auf Constraints eingehen. Eine Regel kann aus einer Kombination von negativen und positiven Applikationen bestehen. Constraints werden in diesem Falle gesondert betrachtet und beziehen sich nicht auf die folgende Funktion. Constraints werden bei den in Kapitel 8 diskutierten Benchmarks nicht berücksichtigt. Daher muss eine Überprüfung dieser Liste an Anwendungsbedingungen durchgeführt werden. Diese Überprüfung erfolgt durch die Funktion „checkACs" in der Package-Datei „MatchConditions.m".

```
Switch[kind, "NAC", newNACList = Append[newNACList,
       {ACList[[j]][[2]],ACList[[j]][[3]]}],
             "PAC", newPACList = Append[newPACList,
```

```
{ACList[[j]][[2]],ACList[[j]][[3]]}]]
```

Getrennt in NACs und PACs wird nun überprüft, ob es einen Widerspruch in der Ausführung der Regel gibt. Sollte die Funktion „checkNACs und die Funktion „checkPACs" den Wert „True" zurückgeben, darf die Transformation nicht durchgeführt werden, wobei in der Funktion „checkPACs" der Wert „True" besagt, daß ein Match zwischen dem Graphen der PAC und dem Startgraphen nicht existiert.

```
If[(checkedN && checkedP) == True,
checked = True,
checked = False];
```

Die eigentlichen Überprüfungen der NACs und PACs erfolgen in den entsprechenden Funktionen, auf die in den beiden nächsten Abschnitten näher eingegangen wird.

7.11.1. Negative Anwendungsbedingungen

Das Überprüfen negativer Anwendungsbedingungen, kurz NACs genannt, erfolgt in der Funktion „checkNACs" in der Datei „MatchCondition.m". Aus Gründen der besseren Übersichtlichkeit wurden Funktionen in Bezug auf Anwendungsbedingungen aus der Klasse „Matching.m" ausgelagert.

Ein getypter Graphmorphismus $m : L \to G$ erfüllt die NAC, wenn kein injektiver, getypter Graphmorphismus $p : NAC \to G$ mit $p \circ x = m$ existiert. Die Funktion mit dem Rückgabewert vom Typ boolean besitzt als Parameter eine Liste aller NACs sowie den Graphen G, die Typisierung passend zum Graphen G und den Morphismus m.

```
checkNACs[NACList_List, G_Graph, typeG_List, m_List]
```

Initialisiert wird die lokale, boolesche Variable mit dem Wert „True". Außerdem wird zu Beginn die Liste des Morphismus m in die beiden Komponenten für Knoten und Kanten aufgeteilt. Die anschließende Do-Schleife gewährleistet die Überprüfung aller negativen Anwendungsbedingungen für die entsprechende Graphtransformationsregel.

Innerhalb des Gültigkeitsbereiches der Schleife wird die aktuelle NAC in die einzelnen Komponenten ähnlich wie beim Vorgehen in Bezug auf den Morphismus m aufgeteilt, wobei hier in einer lokalen Variablen n der aktuelle Morphismus n zwischen der linken Regelseite und dem Graphen der NAC gespeichert wird und die lokalen Variablen NAC und typeNAC als ein Tupel aufgefasst werden. Um für den Morphismus q zwischen dem

Graphen der NAC und dem Graphen G die Werte für die Kanten und Knoten zu initialisieren, werden entsprechend der Größe des Graphen NAC mit Hilfe der Mathematica-Funktion „Table" die Listenpositionen mit der Zahl 0 belegt.

```
{n,{NAC,typeN}}=NACList[[j]];
   {nV, nE} = n;
   nodeAmountNAC = V[NAC];
   edgeAmountNAC = M[NAC];
   qV = Table[0, {nodeAmountNAC}]; (*initialize qV with zeros*)
   qE = Table[0, {edgeAmountNAC}]; (*initialize qE with zeros*)
```

Es muß zudem gezeigt werden, daß das Diagramm $p \circ x = m$ kommutiert. In den beiden For-Schleifen werden nur die Knoten und Kanten in G angesprochen, die einerseits über den Morphismus m und andererseits über die Komposition der Morphismen n und q gleich identifiziert werden. Die Werte für den Morphismus q werden entsprechend den Zuweisungen des bereits festgelegten Morphismus m belegt.

Codefragment für die Knotenbelegung des Morphismus q (für Kanten analog):

```
For[i = 1, i <= Length[nV], i++,
   positionV = i;
   qV[[nV[[i]]]] = mV[[positionV]];
   ];
```

Der Aufruf der Funktion „continueEdgeMatchesSimpleOneMatch" generiert einen Match, der zwischen dem Graphen NAC und dem Graphen G existiert. Gegenüber der Funktion „continueEdgeMatchesSimple", welche alle möglichen Matches bestimmt, wird durch die Berechnung eines einzelnen Matches die Ausführungszeit optimiert. Besteht ein gültiger, injektiver Match, kann die Regel nicht angewandt werden und wird die lokale Variable „match" auf „False" gesetzt und am Ende der Funktion ausgegeben. Ansonsten kann die Regel ausgeführt werden. Das eigentliche Überprüfen der Injektivität erfolgt durch einen Aufruf der Funktion „isInjectiveGraphMorphism" in der Funktion „continueEdgeMatchesSimpleOneMatch". Hierfür muss der entsprechende Parameter in der Funktion „continueEdgeMatchesSimpleOneMatch" auf den Wert „True" gesetzt werden.

```
If[(continueEdgeMatchesSimpleOneMatch[{NAC, typeN},
   {qV, qE},remainingEdgesNAC,{G, typeG},False,{},CompleteGraph[0],True] != {}),
    match = False],
```

In der Notebook-Datei „NACTest.nb" befinden sich mehrere Testbeispiele für die Implementierung der negativen Anwendungsbedingungen.

```
checkNACs[{{nNeu, {NAC, typeNAC}}}, SG, typeSG, m]

True

checkNACs[{{nNeu2, {NAC2, typeNAC2}}}, SG, typeSG, m]

False
```

Abbildung 31: Beispiele der NACs

7.11.2. Positive Anwendungsbedingungen

Eine Graphtransformationsregel kann jedoch auch durch eine Positive Anwendungsbedingung (PAC) eingeschränkt werden und das Ausführen einer Regel nur dann ermöglichen, wenn ein Match zwischen dem Graphen der PAC und dem Graphen G vorhanden ist. Für jede atomare negative Anwendungsbedingung kann, wie in 5.10 gezeigt, eine äquivalente Anwendungsbedingung gefunden werden. Auch in diesem Fall muss das Kommutieren des Diagramms $q \circ x = m$ gezeigt werden. Die Funktion „checkPACs" mit der folgenden Struktur

```
checkPACs[PACList_List, G_Graph, typeG_List, m_List]
```

ist ebenfalls Teil der Package-Datei „MatchConditions.m" und ist nach dem gleichen Prinzip wie die Methode für die NACs aufgebaut.

Zu Beginn erfolgt die komponentenweise Speicherung des Morphismus m in den beiden lokalen Variablen „mV" und „mE" sowie die Initialisierung der booleschen lokalen Variablen „match" mit dem Wert „True". Die aktuelle positive Anwendungsbedingung wird wieder in mehreren Variablen getrennt gespeichert, auf die im weiteren Verlauf der Implementierung analog zur Programmierung der negativen Anwendungsbedingungen zugegriffen werden muss. Zur späteren Generierung der beiden Listen für den Morphismus q zwischen dem Graphen der PAC und dem Hostgraphen initialisiere ich auch hier die einzelnen Listenpositionen, die zuvor durch die Länge der Kantenliste und Knoten im Graphen PAC bestimmt wurden.

```
{n,{PAC,typeN}}=PACList[[j]];
   {nV, nE} = n;
   nodeAmountPAC = V[PAC];
   edgeAmountPAC = M[PAC];
   qV = Table[0, {nodeAmountPAC}]; (*initialize qV with zeros*)
   qE = Table[0, {edgeAmountPAC}]; (*initialize qE with zeros*)
```

Komponentenweise für Knoten und Kanten werden nun die Listenpositionen abgearbeitet und mit Werten belegt. Der Morphismus q wird mit den Werten versehen, die über das gemeinsame Urbild im Graphen L über den Morphismus m auf den Graphen abgebildet werden. Eine notwendige Bedingung ist hier das Kommutieren des Diagrammes $p \circ x = m$. Wir kennen die Realisierung der Implementierung bereits aus dem vorherigen Unterkapitel 7.11.1:

Codefragment für die Knotenbelegung des Morphismus q (für Kanten analog):

```
For[i = 1, i <= Length[nV], i++,
       positionV = i;
   qV[[nV[[i]]]] = mV[[positionV]];
   ];
```

Die vorherige Initialisierung und anschließende Belegung des injektiven Morphismus, sofern er existiert, wenn keine Listenpositionen mit dem Wert 0 am Ende übrigbleiben, erfordert den Aufruf der Funktion „continueEdgeMatchesSimpleOneMatch", der u.a. auch der Morphismus q als Tupel zweier Listen

```
{qV, qE}
```

als Parameter übergeben wird. Aufgrund der Tatsache, daß es sich bei dem Morphismus n zwischen dem Graphen der linken Regelseite und dem Graphen der PAC um eine Inklusion handeln kann, müssen die Kanten und Knoten im Graphen PAC gematched werden, die kein Urbild im Graphen L besitzen. Existiert eine Übereinstimmung in einem Teil des Graphen G, bzw. existiert kein gültiger injektiver Morphismus zwischen dem Graphen PAC und dem Graphen G, muss die entsprechende Regel verworfen werden. Die lokale Variable „match" wird auf den booleschen Wert „False" gesetzt. Letztlich liegt der minimale Unterschied zwischen den beiden zuletzt angesprochenen Funktionen der Anwendungsbedingungen neben einigen Bezeichnungen nur in der Abfrage des Resultates aus der Funktion „continueEdgeMatchesSimpleOneMatch". Das Überprüfen der Injektivität erfolgt ebenfalls durch einen Aufruf der Funktion „isInjectiveGraphMorphism".

7.11.3. Constraints

Gemäß der Definition für Graph-Constraints in 5.12 existiert in der Package-Datei „MatchConditions" die Methode „checkConstraints" für das Überprüfen von Teilgraphen in einem Graph G. Bestimmt werden müssen in erster Linie die beiden Morphismen p und q durch Generierung der Matches, welche mit Hilfe der Funktion „matchesSimple" aus der Datei „Matching.m" errechnet werden. Dies sind alle möglichen Kombinationen möglicher Morphismen zwischen den Graphen P und G:

```
(* all matches for p and q must be created *)
 p = matchesSimple[{P, typeP}, {G, typeG}];
 q = matchesSimple[{C, typeC}, {G, typeG}];
```

Initial befindet sich die lokale Variable „match" in dem Zustand „True". Sollte die Menge p nicht leer sein, q jedoch schon, wird der Wert auf „False" gesetzt.

Für alle generierten Matches von p und q wird in den Do-Schleifen jedes Element der Liste auf Injektivität überprüft. Für diese Abfrage steht uns die Funktion „isInjectiveGraphMorphism" aus der Datei "Morphism.m" zur Verfügung. Ich beschränke mich aufgrund der analogen Struktur der Implementierung bei der visuellen Verdeutlichung des Codes auf die Überprüfung der Elemente von p.

```
Do[
If[isInjectiveGraphMorphism[p[[i]]] == True,
injectiveMatchesP = Append[injectiveMatchesP, p[[i]]]],
   {i, 1, Length[p]}
```

Diese Angaben reichen für die Überprüfung, ob es einen injektiven Match zwischen den Graphen P und G gibt. Trifft dies zu, muss zusätzlich ein injektiver Morphismus q von C nach G existieren. Bei einer Nichtexistenz eines linkseindeutigen Morphismus q wird in der Implementierung der Wert der booleschen Variable „match" auf „False" gesetzt.

```
If[injectiveMatchesP != {},
     If[ injectiveMatchesQ == {},
     match = False,
```

Das Kommutieren des Diagrammes $q \circ a = p$ muss ebenfalls gezeigt werden. Dies erfolgt durch die Hilfsfunktion „checkEquality". Trifft auch diese Eigenschaft auf das entsprechende Beispiel zu, wird Mathematica aus unserem Code den Wert „True" ausgeben und dem Benutzer signalisieren, daß dieser Constraint in diesem Fall erfüllt ist und keine Widersprüche zur Definition vorliegen.

7.12. Priorisierung von Regeln

In der Praxis treten meist Graphgrammatiken mit einer Vielzahl an verschiedenen Regeln auf. Bei einer größeren Menge an Graphtransformationsregeln und einer daraus resultierenden größeren Komplexität der Sequenz von Regeln stellt sich die Frage, ob

man die Regeln entsprechend einer individuell festgelegten oder dem Sachverhalt passenderen Priorisierung deterministisch anwenden kann. Fragen nach einer sequentiellen oder parallelen Unabhängigkeit zwischen den Regeln können hier durchaus eine Rolle spielen. Erst bei einem fehlenden Match einer Regel mit hoher Priorität wird eine Regel mit der nächst niederen Priorität ausgewählt. Um eine bestimmte Reihenfolge der Graphtransformationsregeln festzulegen, existieren mehrere Funktionen in der Klasse „PriorityRules.m", die bspw. die Regeln nach der Anzahl der Knoten und Kanten bestimmen. Weitere Priorisierungs-Funktionen können jederzeit und ohne großen Aufwand für bestimmte Sachverhalte hinzugefügt werden.

Die Funktion „intializeRules" erweitert die Liste der Regeln um einen für die Priorisierung stehenden Wert, wobei die Wertigkeit der Regeln mit der Inkrementierung der Priorisierungsnummer abnimmt. Wir sehen im Folgenden die bereits erwähnte Initialisierungsmethode.

```
initializeRules[r_List] :=
 Module[{initializedList, rangeList},
  rangeList = Range[Length[r]];
  initializedList = Map[(rangeList[[#]] = {r[[#]], #})& , rangeList]
]
```

Ich definiere zuerst eine neue Liste der Länge der als Parameter bereits übergebenen Liste r. Dies erfolgt mit dem von Mathematica angebotenen Listenfunktionals „Range" und wird in der lokalen Variablen „rangeList" zwischengespeichert. Mit der Funktion „Map" definiere ich eine weitere neue Liste, die einerseits mit den Werten von r, andererseits mit den aufsteigenden Listenpositionen belegt wird. Auch in diesem Teil der Implementierung zeigen sich die Vorteile der Übersichtlichkeit und des geringen Codeumfanges in Mathematica. Ich erhalte so in nur zwei Zeilen eine neue Liste der Regeln inklusive einer aufsteigenden Nummerierung.

Im Folgenden in Abbildung 32 sehen wir ein Beispiel der Initialisierungsfunktion:

```
newList = initializeRules[listOfRules]

{{{{-Graph:<1,2,Directed>-, {{1, 1}, {1}}},
   {{1, 2}, {}}, {-Graph:<0,2,Directed>-, {{1, 1}, {}}},
   {{1, 2}, {}}, {-Graph:<1,2,Directed>-, {{1, 1}, {1}}}}, 1},
 {{{-Graph:<1,2,Directed>-, {{1, 1}, {1}}}, {{1, 2}, {}},
   {-Graph:<0,2,Directed>-, {{1, 1}, {}}}, {{1, 2}, {}},
   {-Graph:<1,2,Directed>-, {{1, 1}, {1}}}}, 2}}
```

Abbildung 32: Beispiel der Funktion „initializeRules"

Es ist jedoch ebenfalls möglich, die Prioritäten später individuell anzupassen, da man in

bestimmten Fällen seine vorherige Entscheidung revidieren möchte oder bei einer größeren Anzahl an Regeln lediglich eine geringe Anzahl an Regeln entgegen einer festgelegten Ordnung neu setzen möchte. Die Funktion „setPriority" ersetzt mit Hilfe von „ReplacePart" die Priorisierungsziffer an der entsprechenden Stelle.

```
SetAttributes[setPriority, HoldFirst];

setPriority[r_, pos_Integer, priority_Integer] :=
Module[{},
r =  ReplacePart[r, {pos,2}->priority]
]
```

Wir sehen anhand der Abbildung 33, welche Auswirkung die Funktion „setPriority" auf die übergebene Liste hat. Der Prioritätswert der ersten Regel wird von 1 auf den Wert 5 gesetzt.

```
newSet = setPriority[newList, 1, 5]

{{{{-Graph:<1,2,Directed>-, {{1, 1}, {1}}},
   {{1, 2}, {}}, {-Graph:<0,2,Directed>-, {{1, 1}, {}}},
   {{1, 2}, {}}, {-Graph:<1,2,Directed>-, {{1, 1}, {1}}}}, 5},
 {{{-Graph:<1,2,Directed>-, {{1, 1}, {1}}}, {{1, 2}, {}},
   {-Graph:<0,2,Directed>-, {{1, 1}, {}}}, {{1, 2}, {}},
   {-Graph:<1,2,Directed>-, {{1, 1}, {1}}}}, 2}}
```

Abbildung 33: Beispiel der Funktion „setPriority"

Die Zeile

```
SetAttributes[setPriority, HoldFirst];
```

ermöglicht eine Modifizierung des Wertes der globalen Variablen. Man hätte ohne diese Notation eine Kopie erstellen müssen, die man dann hätte verändern können. „SetAttributes" fügt allgemein ein Attribut einer Liste von Attributen eines bestimmten Symbols zu. „HoldFirst" besagt, daß das erste Argument einer Funktion in einer nichtausgewerteten Form betrachtet wird.

Ich habe drei spezielle Priorisierungsfunktionen als Beispiele implementiert.

```
(* functions for priority-criteria *)

f[r_List] = #1[[2]] < #2[[2]] &
```

```
g[r_List] = V[#1[[1]][[1]][[1]]] <= V[#2[[1]][[1]][[1]]] &

e[r_List] = M[#1[[1]][[1]][[1]]] <= M[#2[[1]][[1]][[1]]] &
```

Die erste Funktion „f" ordnet die Liste der Regeln gemäß ihrer Priorisierung beginnend mit dem kleinsten Priorisierungswert. Die Funktion „g" sortiert die Liste anhand der Anzahl der Knoten in der linken Regelseite, die Funktion „e" anhand der Anzahl der Kanten im Graphen der linken Regelseite. Weitere Sortieralgorithmen können jederzeit ohne großen Aufwand in der Datei „PriorityRules.m" ergänzt werden.

Durch Ausführung von „applyRuleSet" wird eine bestimmte Anzahl an Regelanwendungen lanciert, wobei die Regelmenge aus Regeln mit einer Priorisierung besteht. Der initiale Wert -1 bedeutet, daß eine Restriktion nicht besteht und alle Regeln angewandt werden.

```
If[transformations != -1,
   resultOfXTransf = transform[r,{G,typeG},transformations],
```

In der Funktion „transform" wird nur eine bestimmte Anzahl an Regeln ausgeführt, sofern sie überhaupt durch Abfrage der booleschen Variablen „resultOfTransf" an der Indexstelle 2 den Wert „True" liefert. Zentraler Bereich ist in diesem Codefragment die in der For-Schleife geschachtelte While-Iteration und der Aufruf von „applyArbitraryMatch" aus der Datei „DPO.m".

```
For[j = 2, j <= Length[r], j++,
   While[(counter < max) && (counter <= Length[r]);
    If[counter == 1,
    resultOfTransf = applyArbitraryMatch[newR[[1]], {G, typeG}];
    newGraph = resultOfTransf[[1]],
    resultOfTransf = applyArbitraryMatch[r[[j]][[1]],
    {newGraph[[1]], newGraph[[2]]}];
         If[resultOfTransf[[2]]==True,
       counter++;
       grades = counter;
     Return[{{newGraph, resultOfTransf[[2]]}, grades}]
         ];
      ]
     ]
   ]
```

Die Zählervariable wird durch „counter" repräsentiert und erhöht sich nach jedem erfolgreichen Ausführen einer Graphtransformationsregel. Die Ausgabe dieser Methode besteht aus dem durch die Anwendung der Regeln entstehenden Ergebnisgraphen, einem booleschen Wert zur Kontrolle und der Anzahl der Regelanwendungen.

8. Effizienzanalyse

Im folgenden Kapitel dieser Arbeit führe ich zu Beginn zwei verschiedene Algorithmen und deren Realisierung mit Mathematica ein, um die Effizienz der Implementierung bei gleichzeitiger strikter Berücksichtigung der Graphtransformationstheorie in den Vordergrund zu stellen. Die in Mathematica implementierten Algorithmen Sierpinski (siehe 8.1) und Mutex (siehe 8.2) werden mit den Umsetzungen der beiden Benchmarks in anderen Graphtransformationstools verglichen.

Trotz der Tatsache, daß bereits vor über 30 Jahren das theoretische Konzept der Graphtransformationstheorie fundiert wurde, begann die Entwicklung der Entwicklungsumgebungen für die Graphtransformation weitaus später, befindet sich gar heute noch in einem steten Entwicklungsprozeß. Dabei können Graphtransformationstools, wie wir in Abschnitt 8.4 sehen werden, auf unterschiedliche, funktionale und nichtfunktionale Anforderungen ausgerichtet sein. Relevant für eine Effizienzanalyse sind Aspekte wie Größe bestimmter Muster, maximale Anzahl an Knoten oder die Länge der Regelsequenz.

Ein weiterer Schwerpunkt neben der Einführung der beiden Algorithmen Sierpinski und Mutex ist die Analyse der Zeit, die eine bestimmte Sequenz an Regeln benötigt, auf einem zuvor bestimmten Referenzrechner. Dabei beschränke ich mich auf einen Vergleich mit AGG, wobei über bereits in [$AGT07$], [$BGT05$], [$TR07$] und [$CAGT$] durchgeführte Gegenüberstellungen von AGG zu weiteren Tools Aussagen auch über das Mathematica-Tool getroffen werden können.

8.1. Sierpinski-Benchmark

Ein für die Analyse der Effizienz von Graphtransformationstools geeignetes Algorithmen-Beispiel geht auf das von Waclaw Sierpinski(1882 - 1969) erforschte Sierpinski-Konstrukt zurück. Im Jahre 1915 veröffentlichte der polnische Mathematiker Waclaw Sierpinski, der vor allem durch seine herausragenden Arbeiten im Bereich der Mengenlehre, der Zahlentheorie und Funktionentheorie sowie Topologie bekannt wurde, den Algorithmus namens Sierpinski. Die Implementierung erfordert zwar nur eine geringe Zahl an Regeln, durch ein exponentielles Wachstum eignet sich dieser Benchmark jedoch ideal für eine Vergleichsanalyse.

8.1.1. Sierpinski-Benchmark-Theorie

Ausgangskonstruktion ist ein gleichseitiges Dreieck mit den drei Graphen L (links), R (rechts) und O (oben). Der allgemeine Sierpinski-Algorithmus sieht wie folgt aus:

1. Startzustand in Graph $P = O$. Zufällige Wahl einer der drei Graphen und Verbindung bis zum Mittelpunkt der Geraden zum gewählten Graphen.

2. Endpunkt der Verbindung ist neuer Graph P.

3. Iteration von Punkt 1, bis Rekursionsstufe erreicht ist.

Die Iterationen erfolgen jedoch nicht im inneren, durch die erste Rekursionsstufe entstandenen Dreiecks, daher kann man sich visuell vorstellen, daß das innere Dreieck aus der Konstruktion „herausgeschnitten" wird.

Übertragen wird dieses Algorithmuskonzept auf die Theorie der Graphtransformation, genauer gesagt auf die Struktur von getypten attributierten Graphen. Die Kanten und Knoten der Sierpinski-Graphstruktur sind attributiert in Form von Labels und werden getypt über einen zuvor definierten Typgraphen. An jeder Kante wird in der Mitte ein neuer Knoten hinzugefügt, nur im inneren Dreieck werden keine Verbindungen vollzogen. Die einzelnen Rekursionsstufen werden als Generationen [$AGT07$] bezeichnet. Die Zahl der Knoten beträgt $\frac{3}{2}(1 + 3^n)$ und die Zahl der Kanten ist mit $3^{(n+1)}$ angegeben. Die Zahl n gibt die Generation an. Mit steigender Anzahl der Generationen können die Speicherkapazität sowie die zeitliche Effizienz effektiv getestet werden. Aufgrund dieser Ausrichtung auf große Graphenstrukturen eignet sich dieser Algorithmus besonders für ein Testszenario der Implementierung in Mathematica.

8.1.2. Sierpinski-Benchmark-Implementierung

Ich betrachte in diesem Falle getypte attributierte Graphen. Daher ist die Spezifikation eines attributierten Typgraphen für die weitere Implementierung notwendig. Die Codeumsetzung eines attributierten Typgraphen ist bereits in Teil 7.3 explizit erläutert worden. Dies erfolgt in der Datei „Sierpinski.m", in der alle wichtigen Methoden für den Sierpinski-Algorithmus integriert sind. Das folgende Code-Fragment zeigt die Funktion „typeGraph":

```
typeGraph:=Module[{verticesTG,edgesTG,sourceTG,targetTG,TG},
verticesTG = {"v1"};
edgesTG = {"e12", "e23", "e31"};
sourceTG = {1, 1, 1};
```

```
targetTG = {1, 1, 1};
TG = objectGraph[verticesTG, edgesTG, sourceTG, targetTG]
]
```

Um die bisherigen Angaben für eine Graphgrammatik zu vollenden, benötige ich ebenfalls einen Startgraphen, repräsentiert durch die Methode „startGraph" und die entsprechenden Graphtransformationsregeln, wobei ich bereits konstatiert hatte, daß dieser Algorithmus zur Vergleichsanalyse mit einer geringen Zahl an Regeln auskommt. In dem nächsten Ausschnitt aus der Implementierung sehen wir die bereits erwähnte Funktion „startGraph".

```
startGraph:=Module[{verticesSG,edgesSG,sourceSG,targetSG,SG,typeSG},
verticesSG = {"n1", "n2", "n3"};
edgesSG = {"e12", "e23", "e31"};
sourceSG = {1, 2, 3};
targetSG = {2, 3, 1};
SG = objectGraph[verticesSG, edgesSG, sourceSG, targetSG];
typeSG = {{1, 1, 1}, {1, 2, 3}};
{SG,typeSG}
]


showSierpinsky[level_Integer]:=
Module[{G,typeG},
{G, typeG} = sierpinsky[level];
show[G]

]
```

Eigentlich besteht dieses Konzept aus lediglich einer Regel, die rekursiv auf die entsprechenden Teilgraphen angewandt werden kann. Die linke Seite wird durch Aufruf der Funktion „startGraph" generiert. Der Klebegraph besteht in der Regel aus drei Knoten der gleichen Typisierung, während zwischen den Knoten keine Kanten existieren. Diese Graphtransformationsregel löscht demnach die Verbindungen zwischen den Knoten. Die rechte Regelseite enthält zusätzlich drei Knoten der gleichen Typisierung und fügt Kanten eines unterschiedlichen Types in den Graphen ein, sodaß sich drei neue Dreiecke in dem alten Dreieck bilden. In der Methode „rule" wird die Definition der Regel für unser Sierpinski-Benchmark realisiert.

Je nachdem, ob der Schwerpunkt bei der zeitlichen Effizienz oder bei der strikten Orientierung an den theoretischen Grundlagen liegt, kann man zwischen zwei Sierpinski-Funktionen wählen. Im Folgenden sehen wir die Standard-Funktion:

```
sierpinsky[level_Integer]:=
Module[{x,L,typeL,G,typeG,matches,parallelRule,parallelMatch},

{G,typeG}=startGraph;
{L, typeL}=rule[[1]];


(* apply parallel rule for all matches at each level *)
Do[
matches = matchesSimple[{L, typeL}, {G, typeG}];
{parallelRule, parallelMatch} =
   parallelProduction[rule,matches];
{G, typeG}=applyRule[parallelRule,parallelMatch,{G, typeG}]
,{i,level}
];

{G,typeG}
]
```

Der Funktion „sierpinski" wird ein Integer-Wert, repräsentativ für den Wert der Generation oder Rekursionsstufe, übergeben. Initialisiert werden zu Beginn die beiden Graphen der linken Seite sowie der Graph G durch Aufruf der Methode „startGraph" bzw. „rule". Die Iteration durch ein Do-While-Konstrukt ist abhängig von der Zahl der Generation. In jedem Schritt werden zuerst die Matches bestimmt, mit Hilfe der Matches als Parameter für die Funktion „parallelProduction" die Regelanwendung parallelisiert und der Zielgraph generiert.

Desweiteren steht die optimierte Methode „sierpinskyO" zur Verfügung. Der Unterschied besteht darin, daß anstatt der Methode „applyRule" die optimierte Funktion „applyRuleO" verwendet wird, in der bspw. auf die Generierung der beiden Morphismen n und g bei der Graphtransformation verzichtet wird, was zeitliche Vorteile dadurch mit sich bringt. Für das Ergebnis der Transformation werden die beiden Morphismen n und g nicht benötigt. Um die in Abbildung 34 gezeigte Funktion zu erläutern, möchte ich kurz auf einige zentrale Code-Zeilen eingehen. Beginnend mit der Generierung des Startgraphen und der Regel erfolgt die Bestimmung der Anzahl der Anwendung der Regel gemäß der individuellen Wahl der Rekursionsstufe. In der Do-Schleife werden zu Beginn die Matches zwischen dem linken Graphen der Regel und dem Graphen G bestimmt, anschließend parallel die Regel auf alle ermittelten Matches angewandt. Der Ergebnisgraph wird nach der Parallelisierung der Regelanwendung mitsamt der Typisierung ausgegeben. Die Parallelisierung ist in diesem Fall möglich, da in diesem Beispiel alle Matches automatisch voneinander abhängig sind.

Eine Visualisierung der Implementierung erfolgt in den dazugehörigen Notebook-Dateien, in denen der Code auf Fehler überprüft werden kann. In der Datei „SierpinskiVisual.nb" werden inkrimentierend die einzelnen Rekursionsstufen überprüft. Anbei ist die Graphik 34 der ersten Rekursionsstufen.

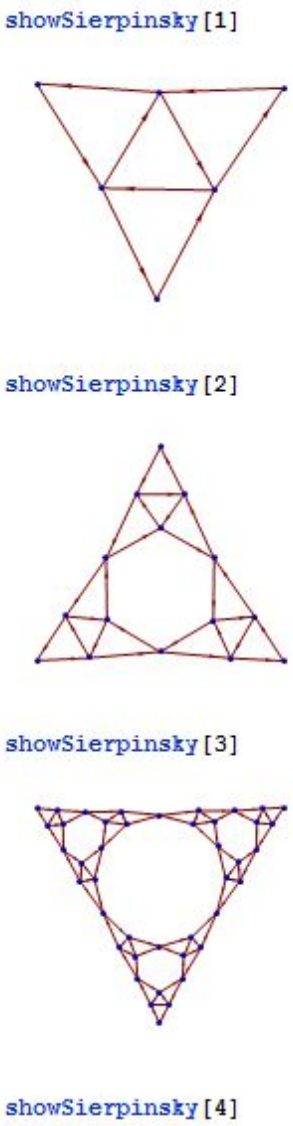

Abbildung 34: Rekursionsstufen der Sierpinski-Implementierung

Die Visualisierung der sechsten Rekursionsstufe sieht wie folgt aus:

```
showSierpinsky[6]
```

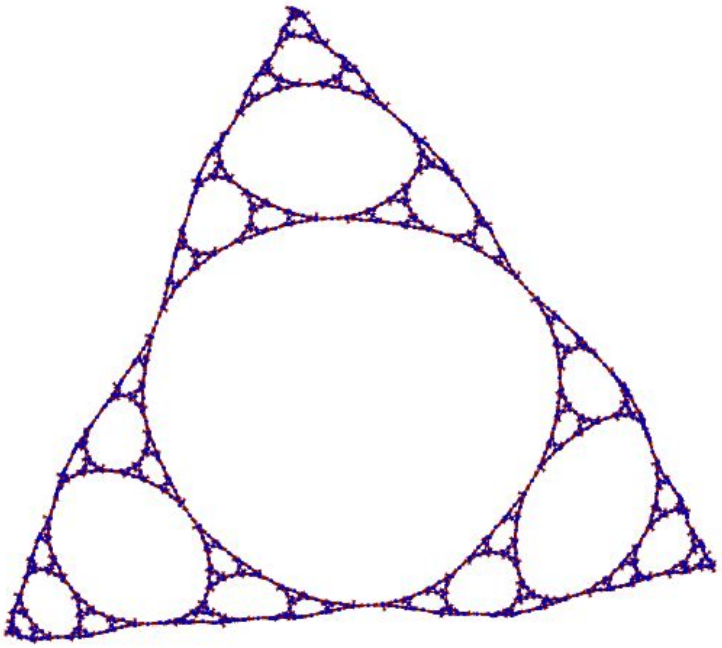

Abbildung 35: Rekursionsstufe 6 der Sierpinski-Implementierung

8.1.3. Ergebnis der Vergleichsanalyse

Eine aussagekräftige Vergleichsanalyse der Mathematica-Implementierung zur Realisierung in AGG erfordert die Festlegung eines Referenzrechners. Die Tests in Bezug auf die Sierpinski- und Mutex-Implementierung wurden auf dem folgenden System durchgeführt:

*Dell Inspiron M*1710
*Intel(R) Core(TM) 2 CPU*7400 ;2, 16 *GHZ*
RAM : *2*046 *MB*
Windows Vista

Um ebenfalls unter fairen Bedingungen eine Effizienzanalyse durchzuführen, wurden die Implementierungen des Sierpinski-Algorithmus jeweils unter der AGG-GUI und der Mathematica-GUI getestet. Ein frappierender Unterschied zwischen einer Testumgebung

in der Konsole und der graphischen Benutzeroberfläche wurde vor allem bei AGG sichtbar. Die Zeitwerte schwankten von ca. 10 ms in der Konsole, die ich über Eclipse gestartet hatte, bis zu 269 000 ms in der AGG-GUI.

Um eine vergleichbare Ausführungszeit der Regelanwendungen unter Mathematica zu erhalten, habe ich mich für die von Mathematica in der umfangreichen Bibliothek bereitgestellte Methode „Timing" entschieden, die bereits in Kapitel 6.5 vorgestellt wurde.

```
{time, {G7, typeG7}} = Timing[sierpinskyO[7]]; time
75.723
```

Abbildung 36: Zeitmessung der Funktion sierpinskyO

Wir sehen in Abbildung 36 eine Zeitmessung der optimierten Variante der Sierpinski-Berechnung.

Trotz der Tatsache, daß die Mathematica-Implementierung innerhalb kürzester Zeit entwickelt wurde, zeigt vor allem das Ergebnis bezüglich eines Vergleiches in der GUI, daß hier bereits das Graphtransformationstool unter Mathematica bei der Generation 7 mit ca. 77 Sekunden weitaus schneller als das AGG-Tool agiert, welches ca. 264 Sekunden für die Berechnungen benötigt.

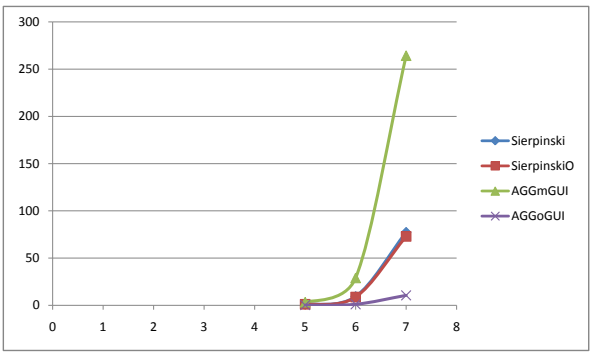

Abbildung 37: Diagramm der Vergleichsanalyse

Das Diagramm in Abbildung 37 gibt einen visuellen Einblick in die Vergleichsanalyse. An der X-Achse werden die Zeitwerte in Sekunden angegeben und an der Y-Achse der Wert der Generation. Der blaue Graph symbolisiert die Ergebnisse der Standard-Implementierung des Sierpinski-Algorithmus in Mathematica. Die rote Markierung bezieht sich auf die Zeitwerte für die optimierte Variante der Implementierung in Mathematica. Die Abkürzung „AGGmGUI" in der Legende neben dem Diagramm steht für

das Testen der AGG-Implementierung mit graphischer Benutzeroberfläche, während die Abkürzung „AGGoGUI" für die Anwendung des AGG-Tools ohne Aktivierung der GUI zu verstehen ist.

Die „Profile"-Funktionalität bietet dem Benutzer der Software Mathematica die Möglichkeit der detaillierten Überprüfung der Ausführungszeit der einzelnen Aufrufe einer zu untersuchenden Funktion, um herauszufinden, an welchen Stellen weitere Optimierungen denkbar wären. Diese Operation kann lediglich mit zusätzlicher Zeitverzögerung im Debugging-Modus durchgeführt werden und erstellt automatisch eine neue Datei mit den gewünschten Ausgabewerten. Der größte zeitliche Aufwand erfolgt durch das Matching zwischen dem Graphen der linken Seite und dem Graphen, auf den die Graphtransformationsregel angewandt werden soll. In der Abbildung 38 ist eine solche Detailansicht der Ausführungszeit für den Sierpinski-Aufbau der Generationsstufe 5 zu erkennen.

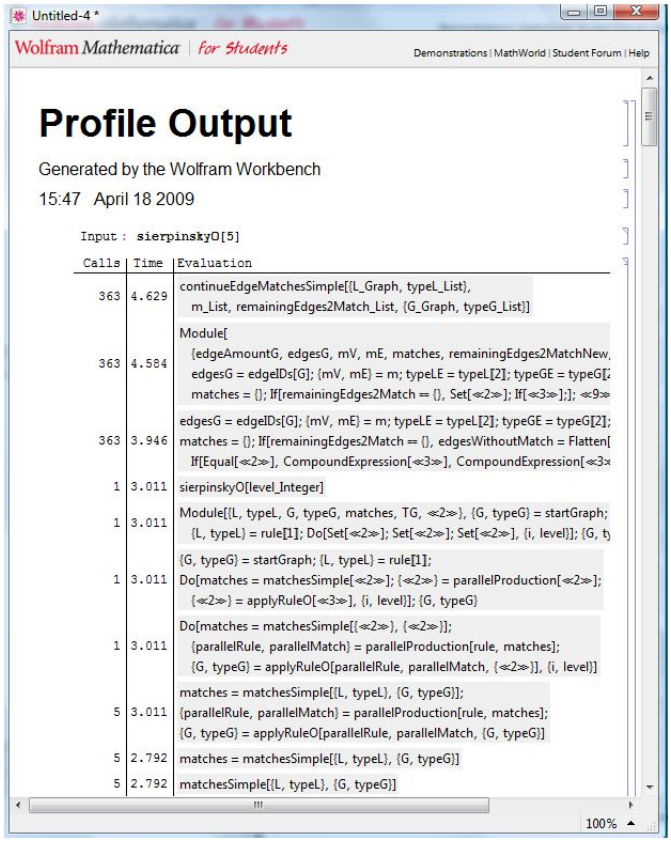

Abbildung 38: Profile der Funktion „sierpinsky" der Generation 5

In der linken Spalte mit der Bezeichnung „Calls" werden die Aufrufe der einzelnen Codezeilen bestimmt. In der zweiten Zeile „Time" erfolgt die genaue Zeitangabe und in der dritten Spalte ist angeben, welche Codezeile evaluiert wurde.

8.2. Mutex-Benchmark

Ein weiterer Algorithmus, der im Bereich der Vergleichsanalyse bei Graphtransformationstools eine wesentliche Rolle spielt und als Maßstab für eine effiziente Umsetzung der Implementierung dient, ist der Mutual Exclusion-Algorithmus. Wir werden in den folgenden Abschnitten zuerst eine ausführliche Schilderung der Graphtransformationsregeln und der semantischen Abläufe sehen. Es folgt eine Erläuterung der Realisierung der Regeln in Mathematica. Der Typgraph, der das Metamodell beschreibt, wird im folgenden Abschnitt 8.2.1 durch die Abbildung 39 illustriert und besteht lediglich aus zwei Klassen (process und resource). Dieser Algorithmus beschreibt die Problematik des Zugriffes mehrerer Prozesse auf eine Ressource mit der Einschränkung, daß auf jede Ressource zu einem Zeitpunkt nur von einem Prozeß zugegriffen werden kann. Das Recht, auf eine Ressource zuzugreifen, wird durch ein Token, in diesem Falle durch eine Kante vom Typ „token", symbolisiert. Ein Prozeß kann durch eine Anfrage überprüfen, ob eine bestimmte Ressource belegt oder frei ist. Ebenfalls kann eine Ressource durch eine entsprechende Regelanwendung wieder freigegeben werden. Der Algorithmus überprüft das Vorhandensein von deadlocks, indem er alle Prozesse einzeln betrachtet und überprüft, ob sie blockiert sind.

8.2.1. Mutex-Benchmark-Theorie

Der Mutual Exclusion-Algorithmus, der grundsätzlich in einer visuellen Sprache mit dynamischer Semantik definiert wird, kann durch 13 Regeln beschrieben werden, auf die ich im Verlauf dieses Abschnitts näher eingehen werde und die im Abschnitt 8.2.2 als Code angeführt werden. Die Regelanwendungen sollen ein mögliches Verhalten des Systems in unterschiedlichen Situationen beschreiben. Die Knotenmenge besteht lediglich aus Knoten vom Typ „Process" oder „Resource". Die komplexeste Regel namens „blockedRule" beinhaltet 4 Knoten und 3 Kanten. Prädestiniert für eine Effizienzanalyse sind Mutex-Graphen mit einer hohen Anzahl an Knoten vom Typ „Process". Die Abbildung 39 zeigt den Typgraphen des Mutex-Algorithmus, über den die Graphen der Regel getypt werden. Die für die Sequenz Mutex relevanten Regeln werden in Abschnitt 80 beschrieben.

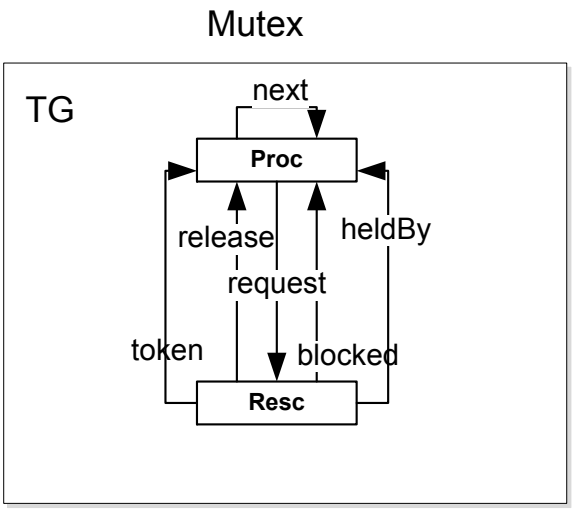

Abbildung 39: Typgraph Mutex

Die Regel „newRule" fügt einem Graphen einen Prozeßknoten hinzu. Die erzeugende Regel „mountRule" (siehe Abbildung 40) fügt einem Prozeßknoten eine Kante des Typs „token" hinzu und verbindet den Knoten mit einem Resource-Knoten.

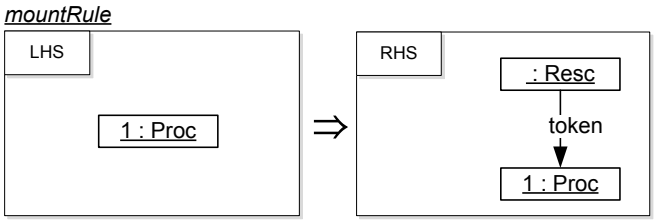

Abbildung 40: Regel mountRule

Die Regel „requestRule", illustriert durch Abbildung 41, fügt zwischen einem Prozeßknoten und einem Resource-Knoten eine Kante mit dem Typen „request" hinzu. Berücksichtigt werden die ersichtlichen drei negativen Anwendungsbedingungen. Sollten diese Subgraphen Teil des Graphen G sein, ist die Regel an dem jeweiligen Match nicht anwendbar.

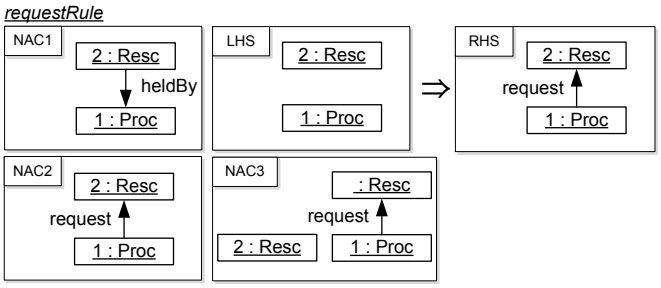

Abbildung 41: Regel requestRule

Die Graphtransformationsregel „takeRule" (siehe 42) beschreibt den Zugriff eines Prozesses auf eine Ressource nach einer Anfrage. Die Kante „token", die signalisiert, daß der entsprechende Prozeß ein Zugriffsrecht auf diese Ressource besitzt, wird durch die Regel gelöscht. Die hinzugefügte Kante vom Typ „heldBy" symbolisiert den Zugriff auf die Ressource.

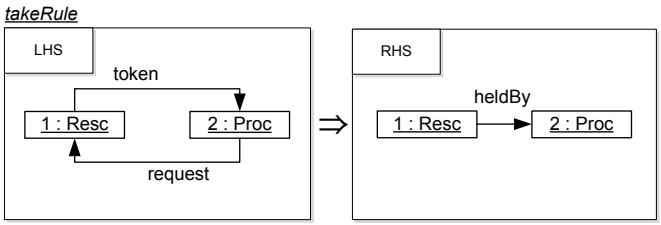

Abbildung 42: Regel takeRule

Durch Anwendung der Regel „releaseRule" (siehe 43), bei der ebenfalls eine negative Anwendungsbedingung berücksichtigt werden muss, wird eine Kante mit dem Typen „heldby" entfernt und eine Kante des Typs „release" hinzugefügt. Diese Regel beschreibt die Freigabe einer Ressource.

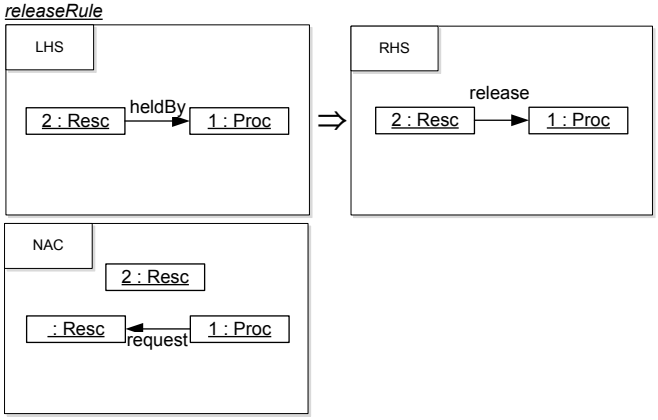

Abbildung 43: Regel releaseRule

Die letzte Regel (siehe 44) der ersten Sequenzvariante enthält in der linken Regelseite zwei Prozeßknoten und einen Resource-Knoten. Zwischen dem ersten Prozeßknoten und dem Resource-Knoten besteht eine release-Kante und zwischen den beiden Prozeßknoten eine Next-Kante. Während die Next-Kante durch die Regel bestehen bleibt, wird die Release-Kante gelöscht. Zusätzlich existiert in der rechten Regelseite eine Kantenverbindung des Typs „token" zwischen dem zweiten Prozeßknoten und dem Resource-Knoten. Die Regel beschreibt die Gewährung eines Zugriffsrechtes einer gerade freigegebenen Ressource.

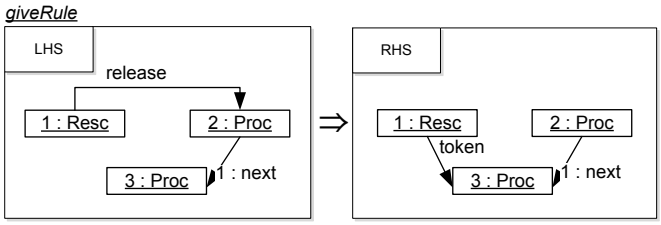

Abbildung 44: Regel giveRule

Ich komme zu den Regeln für die Sequenz MutexStar gemäß der Notation in [$TR07$]. Die Reihenfolge der Regelanwendungen wird in Abschnitt 8.2.2 erläutert.

Auch hier erfolgt zu Beginn der Aufruf der Regel „newRule", um zwischen zwei Prozeßknoten einen weiteren Prozeßknoten einzufügen. Es folgt ein Aufruf der Graphtransformationsregel „attachResource" (siehe 45). Einem Prozeßknoten wird eine Kantenverbindung des Typs „heldby" hinzugefügt, die den Prozeßknoten mit einem ebenfalls hinzugefügten Resource-Knoten verknüpft. Beachtet werden muss in diesem Fall eine negative Anwendungsbedingung. Die NAC gewährleistet, daß der entsprechende Prozeßknoten keine bereits vorhande Kantenverbindung „heldby" zu einem anderen Resource-Knoten besitzt. Einem Prozeß wird direkt ohne Anfrage der Zufriff auf eine Ressource gewährt.

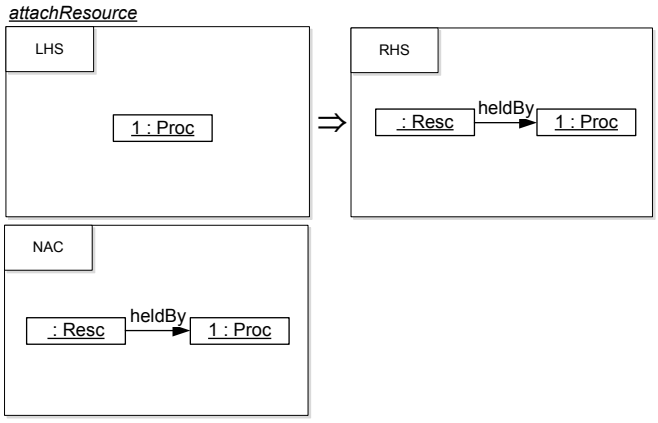

Abbildung 45: Regel attachResource

Die Regel „requestStarRule" (siehe 46) besitzt in der linken Regelseite zwei Prozeßknoten und zwei Resource-Knoten. Die beiden Prozeßknoten sind untereinander über eine Next-Kante und über eine heldby-Kante mit jeweils einem Resource-Knoten verbunden. Die Regel ist eine erzeugende Regel, indem lediglich zwischen dem ersten Prozeßknoten und dem zweiten Resource-Knoten eine neue Kante vom Typ „request" hinzugefügt wird. Diese Regel besteht ebenfalls aus einer NAC, die besagt, daß zwischen dem ersten Prozeßknoten und dem zweiten Resource-Knoten keine Kante des Typs „request" bestehen darf.

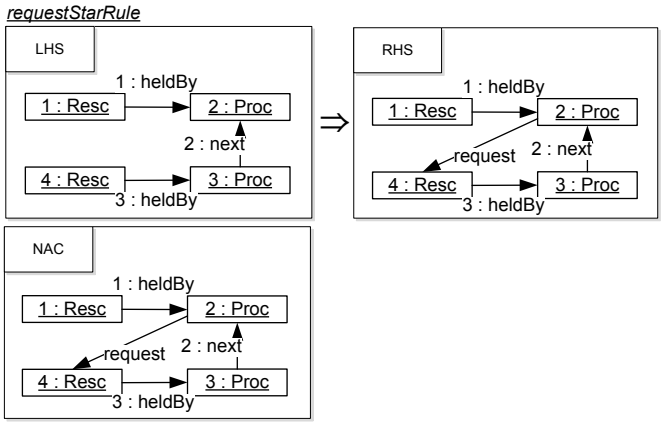

Abbildung 46: Regel requestStarRule

Die Regel „blockedRule" (siehe 47) besteht aus zwei Prozeßknoten und einem Resource-Knoten. Eine Kantenverbindung mit dem Typ „request" verbindet den ersten Prozeßknoten mit dem Resource-Knoten. Zwischen dem Resource-Knoten und dem Prozeßknoten existiert eine heldby-Kante. Die Regel fügt eine blocked-Kante zwischen dem Resource-Knoten und dem ersten Prozeßknoten hinzu. Die Graphtransformationsregel „blockedRu-

le" signalisiert dem anfragenden Prozeß, daß die gewünschte Ressource blockiert ist.

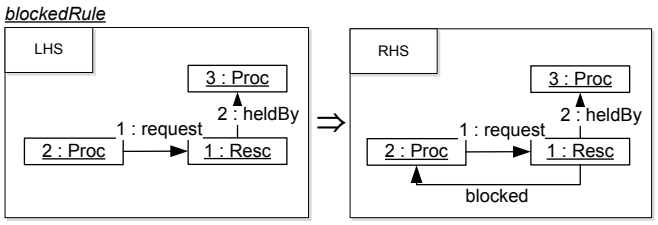

Abbildung 47: Regel blockedRule

Die aus zwei Prozeßknoten und zwei Resource-Knoten bestehende Regel „waitingRule"(siehe 48) löscht die blocked-Kante zwischen dem ersten Resource-Knoten und dem ersten Prozeß und fügt eine blocked-Kante an den zweiten Prozeß an. Die beiden Kanten des Typs „heldby" und „request" werden durch Anwendung der Graphtransformationsregel erhalten.

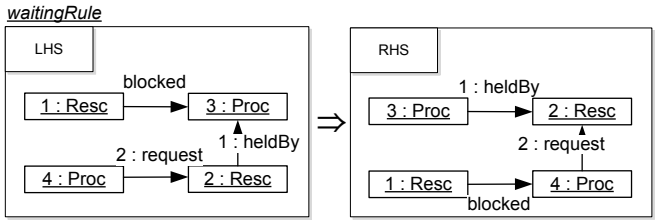

Abbildung 48: Regel waitingRule

Es folgt die Regel „unlockRule" (siehe 49). Der Graph der linken Regelseite besteht aus einem Prozeß und einem Resource-Knoten, die beide mit den beiden Kanten „blocked" und „heldby" verbunden sind. Die Regel löscht beide Kantenverbindungen und fügt eine Kante „release" hinzu. Die Regel beschreibt die Freigabe einer blockierten Ressource.

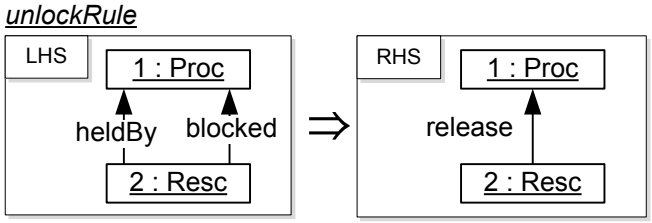

Abbildung 49: Regel unlockRule

Die folgende Regel heißt „ignoreRule" (siehe 50) und entfernt eine blocked-Kante zwischen einem Resource- und einem Prozeßknnoten. Berücksichtigt werden muss in diesem Fall eine negative Anwendungsbedingung. Der Prozeß darf keine Verbindung des Typs „heldby" zu einem Resource-Knoten besitzen.

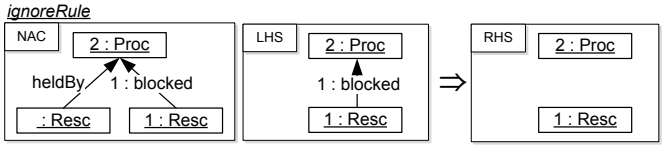

Abbildung 50: Regel ignoreRule

Die folgende Regel ist zwar aus der ersten Regelsequenz bekannt, dennoch werde ich die Regel erneut kurz beschreiben. Sie wird in der zweiten Regelsequenz erneut benötigt, um einem Prozeß wieder das Zugriffsrecht auf eine entsprechende Ressource zu gewähren. Die Graphtransformationsregel „giveRule" (siehe 51) besitzt in der linken Regelseite zwei Prozeßknoten, die durch eine next-Kante miteinander verbunden sind und durch einen Resource-Knoten. Zwischen dem Resource-Knoten und dem ersten Prozeßknoten besteht zudem eine Kante des Typs „release". Die Regel löscht die release-Kante und fügt zwischen dem Resource-Knoten und dem zweiten Prozeßknoten eine Kante des Typs „token" hinzu.

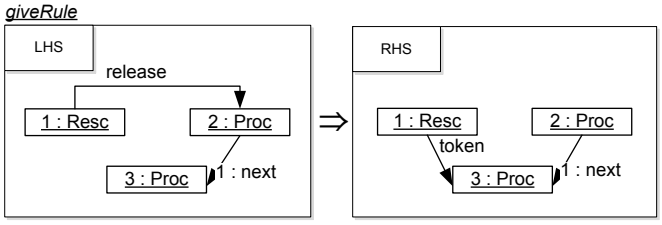

Abbildung 51: Regel giveRule

Die vorletzte Regel namens „takeRule" (siehe 52) löscht zwei Kanten des Typs „token" und „request" zwischen einem Resource-Knoten und einem Prozeß und fügt eine heldby-Kante hinzu.

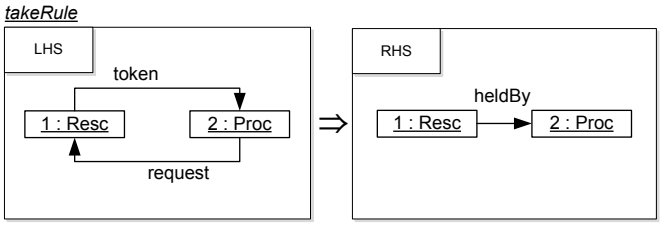

Abbildung 52: Regel takeRule

Die Regel „releaseStarRule" (siehe 53) besitzt in der linken Seite zwei Prozeßknoten und zwei Resource-Knoten. Der erste Prozeß ist mit dem ersten Resource-Knoten über eine request-Kante verbunden. Beide Resource-Knoten sind zudem mit dem zweiten Prozeßknoten über heldby-Kanten verbunden.

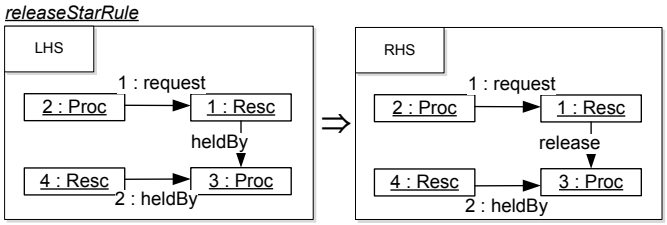

Abbildung 53: Regel releaseStar

8.2.2. Mutex-Benchmark-Implementierung

Die grundlegende Funktion „mutex" befindet sich in der gleichnamigen Datei „Mutex.m" und verarbeitet die definierten Regeln, wendet die Regeln an und generiert den Ergebnisgraphen. Zu berücksichtigen ist in diesem Fall im Gegensatz zur Implementierung des Sierpinski-Algorithmus die Existenz von negativen Anwendungsbedingungen. Analog zur Sierpinski-Umsetzung in Mathematica definiere ich hier ebenfalls einen Startgraphen und einen Typgraphen. Um benutzerfreundlich die Generierung beliebig großer Mutex-Graphen zu gewährleisten, kann mit der Funktion „createGraph" durch eine Schleifenkonstruktion die Anzahl der Prozeßknoten festgelegt werden. Ein Match, der in jeder Iterationsstufe der Schleife neu berechnet werden muß, reicht aus. Daher erfolgt hier der Aufruf von "continueEdgeMatchesSimpleOneMatch". Nun folgt eine genauere Erläuterung der Methode „mutex".

Der Funktion werden der Graph G, die dazugehörige Typisierung sowie die entsprechende Regel als Parameter übergeben. Zu Beginn erfolgen für den weiteren Verlauf des Codes notwendige Initialisierungen. Zur Bestimmung der linken Regelseite inklusive der Typisierung werden mit dem Index 1 der Liste „whichRule" die notwendigen Werte herausgelesen. Anschließend trennt man die entsprechende Regel von den Anwendungsbedingungen, die in der lokalen Variablen „ruleWithoutNAC" gespeichert werden. Die negativen Anwendungsbedingungen, falls welche vorhanden sind, speichert man in der Variablen „NAC".

```
{L, typeL} = whichRule[[1]];
ruleWithoutNAC = Delete[whichRule, 6];
NAC = whichRule[[6]];
```

Es folgt eine Überprüfung der Existenz von Anwendungsbedingungen. Existieren keine Anwendungsbedingungen, kann die ausgewählte Regel parallelisiert werden, indem die Funktion „parallelProduction" (siehe 7.9) ausgeführt wird. Die Ermittlung der Matches muss zwingend in beiden Fällen durchgeführt werden. Existiert mindestens eine NAC wird der zweite If-Zweig erreicht und die sich ebenfalls in dem gleichen Paket befindliche Methode „parallelWithACs" aufgerufen. Ein Aufruf von „parallelProduction" darf hier

nicht erfolgen, denn die Regeln können voneinander abhängig sein und bei einer parallelen Anwendung zu Konflikten führen. Wir sehen anschließend die gerade erläuterte Schleifenkonstruktion:

```
(* check, if the rule contains a NAC *)
If[First[NAC]=={},
   (* newG = applyArbitraryMatch[ruleWithoutNAC, {G, typeG}] *)
   matches = matchesSimple[{L, typeL}, {G, typeG}];
   {parallelRule, parallelMatch} =
   parallelProduction[ruleWithoutNAC,matches];
      newG = applyRule0[parallelRule,parallelMatch,{G, typeG}]
   (*newG = applyRule[parallelRule,parallelMatch,{G, typeG}]*)
,

      matches = matchesSimple[{L, typeL}, {G, typeG}];
      newG = parallelWithACs[ruleWithoutNAC, matches, NAC, {G, typeG}]
  ]
```

Eine bestimmte Regelsequenz kann jedoch auch erforden, daß eine Regel nicht parallel, sondern nur ein einziges Mal oder sequentiell aufgerufen werden soll. In diesem Fall wird nicht die bereits zuvor beschriebene Funktion „mutex" ausgewählt, sondern die Methode „singleMutex", welche ebenfalls den Startgraphen, dessen Typisierung und die Regel als Parameter übergeben bekommt.

Die Abfrage bezüglich der Existenz einer Anwendungsbedingung wird erneut durch eine If-Konstruktion gewährleistet. Enthält die als Parameter übergebene Funktion keine Anwendungsbedingung, wird durch die Funktion „continueEdgeMatchesSimpleOneMatch" ein gültiger Match bestimmt. Zuvor müssen jedoch verschiedene Initialisierungen durchgeführt werden, um passende Parameter für das Matching bereit zu stellen. Die Überprüfung der Gluing-Condition erfolgt erst durch „applyRuleO", daher wird in dieser Funktion bereits in „continueEdgeMatchesSimpleOneMatch" überprüft, ob der generierte Match die Gluing-Condition erfüllt. Die boolesche Variable „CheckGC" muss in diesem Fall auf den Wert „True" gesetzt werden.

```
{L, typeL} = whichRule[[1]];
l = whichRule[[2]];
{K, typeK} = whichRule[[3]];
CheckGC=True;
ruleWithoutNAC = Delete[whichRule, 6];
NAC = whichRule[[6]];
remainingEdges = {};
nodeAmount = V[L];
```

```
edgeAmount = M[L];
mV = Table[0, {nodeAmount}]; (*initialize qV with zeros*)
mE = Table[0, {edgeAmount}]; (*initialize qE with zeros*)
```

Um eine Termination der Regelsequenz vor Aufruf der letzten Regel zu verhindern, wird bei keinem gültigen Match der unveränderte Startgraphen ausgegeben. Dies wird durch die Abfrage

```
If[match == {},
  newG = {G,typeG};
  Break[]
  ];
```

gelöst. Diese Umsetzung widerspricht zwar dem theoretischen Konzept einer Regelsequenz in der Graphtransformationstheorie, ist jedoch erforderlich, um einen authentischen Vergleich zu AGG durchführen zu können. In AGG wird bei einem Konflikt mit einer Regel die Sequenz nicht abgebrochen, sondern die nächste Regel untersucht.Die Werte der Häufigkeit einer Anwendung einer Regel sind in den Einstellungen von AGG lediglich Maximalwerte.

Bei einer Existenz einer Anwendungsbedingung, im Falle dieses Benchmarks lassen sich nur negative Anwendungsbedingungen finden, müssen die Matches durch Aufruf der Funktion „checkACs" auf Konflikte überprüft werden. Ist der erste Match aufgrund der NAC kein gültiger Match, wird der nächste Match in der Do-Schleife überprüft.

Neben der Möglichkeit der Anwendung einer der definierten Regeln existiert die Möglichkeit eine Funktion auszuwählen, die eine Ausführung einer Sequenz von Regeln gemäß den Angaben bei AGG zur Folge hat. Diese Methode namens „sequenceMutex" befindet sich ebenfalls in der Datei „Mutex.m". Beginnend mit der Generierung der Anzahl der Prozesse, genauer gesagt der Prozeßknoten, wird in der für die Analyse ausgewählten Sequenz im nächsten Schritt einmal die Regel „mountRule" angewandt. Es folgt die Anwendung der Regel „requestRule", bis die Regel nicht mehr angewandt werden kann. Die for-Schleife in der Implementierung beschreibt das hundertmalige Anwenden der Dreierregelsequenz 1.takeRule, 2.releaseRule, 3.giveRule. Am Ende wird der Ergebnsigraph in der Variablen „newGraph" gespeichert und ausgegeben. Wir sehen im Folgenden die Implementierung dieser Methode mit dem Namen „sequenceMutex":

```
(* size of benchmarks is chosen in the notebook-file *)
sequenceMutex[n_Integer]:=Module[{newGraph
},
newGraph=createGraph[n];
newGraph=singleMutex[newGraph, mountRule];
```

```
newGraph=mutex[newGraph, requestRule];
For[i = 0, i <n, i++,
 newGraph=singleMutex[newGraph, takeRule];
 newGraph=singleMutex[newGraph, releaseRule];
 newGraph=singleMutex[newGraph, giveRule]
   ];
newGraph
]
```

Der dazugehörige Ergebnisgraph der Mutex-Sequenz der Prozeßknotengröße 10 wird durch Abbildung 54 illustriert. Es ist ein Auszug aus der Datei „Mutex.nb". In der Variablen „sequenceGraph" wird der Ergebnisgraph und die dazugehörige Typisierung gespeichert. Angezeigt werden die Anzahl der Knoten, die Anzahl der Kanten, die Angabe, ob es sich um gerichtete oder ungerichtete Kanten handelt sowie die Typisierung, die in eine Liste der Knotentypisierungen und Kantentypisierungen aufgeteilt wird.

```
(* ********************** Sequence ************************** *)

sequenceGraph = sequenceMutex[10]

{- Graph:< 11,11,Directed >-,
 {{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2}, {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3}}}

show[sequenceGraph[[1]]]
```

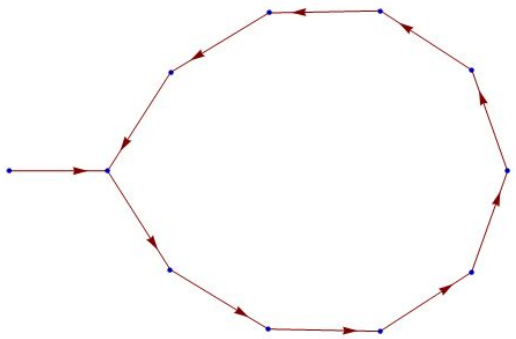

Abbildung 54: Visualisierung des Ergebnisgraphen der Mutex10-Sequenz

Abbildung 55 zeigt den Ergebnisgraphen der Mutex100-Sequenz.

```
(* ************************ Sequence **************************** *)

In[17]:= sequenceGraph = sequenceMutex[100]

Out[17]= {- Graph:< 101,101,Directed >-,
          {{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
            1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
            1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
            1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2},
           {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
            1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
            1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
            1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3}}}

In[18]:= show[sequenceGraph[[1]]]
```

Out[18]=

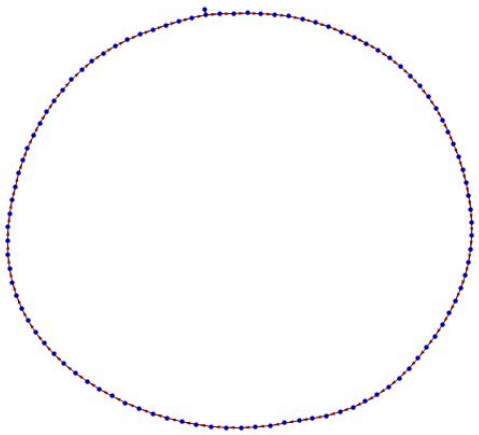

Abbildung 55: Visualisierung des Ergebnisgraphen der Mutex100-Sequenz

Eine zweite Sequenzvariante wird „MutexStar" genannt. Auch in diesem Fall wird zuerst die Anzahl der Prozeßknoten durch „createGraph" bestimmt. Zu beachten ist das Vorkommen der vier For-Schleifen. Beendet wird diese Regelsequenz mit dem einmaligen Aufrufen der Funktionen „giveRule" und „takeRule" und mit der Ausgabe des Ergebnisgraphen, der in der lokalen Variablen „newGraph" gespeichert wird.

```
(* size of benchmarks is chosen in the notebook-file *)
sequenceMutexStar[n_Integer]:=Module[{newGraph
},
newGraph=createGraph[n];
newGraph=mutex[newGraph, attachResource];
For[i = 0, i <n, i++,
 newGraph=mutex[newGraph, requestStarRule];
 ];
```

```
newGraph=singleMutex[newGraph, blockedRule];
For[i = 0, i <n-1, i++,
 newGraph=mutex[newGraph, waitingRule];
];
newGraph=singleMutex[newGraph, unlockRule];
newGraph=singleMutex[newGraph, bockedRule];
For[i = 0, i <n-1, i++,
 newGraph=mutex[newGraph, waitingRule];
];
newGraph=singleMutex[newGraph, ignoreRule];
For[i = 0, i <n-1, i++,
 newGraph=singleMutex[newGraph, giveRule];
 newGraph=singleMutex[newGraph, takeRule];
 newGraph=singleMutex[newGraph, releaseStarRule]
 ];
newGraph=singleMutex[newGraph, giveRule];
newGraph=singleMutex[newGraph, takeRule]
]
```

Diese Regelsequenz orientiert sich an der Abfolge der Regelanwendungen der AGG-Einstellungen. Selbstverständlich kann auch eine Sequenz gemäß dem ALAP(As Long As Possible)-Prinzips angegeben und der Ergebnisgraph berechnet werden. Die für diesen Benchmark nötigen Funktionen zur Berechnung der Matches, zur Regelanwendung und zum Überprüfen der Anwendungsbedigungen, auf die ich im vorherigen Kapitel 7 explizit eingegangen bin, bilden auch hier die Grundlage. Eine Änderung der Regelsequenz würde in der Package-Datei „Mutex.m" erfolgen.

Abbildung 56 zeigt einen Aufruf der Regelsequenz MutexStar und Visualisierung des Ergebnisgraphen für zehn Prozeßknoten.

8.2.3. Ergebnis der Vergleichsanalyse

Ich beschränke mich bei der Vergleichsanalyse der standardmäßigen Mutex-Sequenz auf Graphen der Prozessknotengrößen 10 und 100. Trotz der Kompaktheit der Implementierung unter Mathematica liegen die Ergebnisse der Zeitanalyse im Bereich der AGG-Realisierung. Ich komme zuerst zur Auswertung der standardmäßigen Mutex-Sequenz.

```
(* the Mutex-sequence ''Star'' according the AGG-implementation *)

In[3]:= sequenceGraph2 = sequenceMutexStar[10]

Out[16]= {- Graph:< 20,20,Directed >-,
          {{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2},
           {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2}}}

In[17]:= show[sequenceGraph2[[1]]]
```

Out[17]=

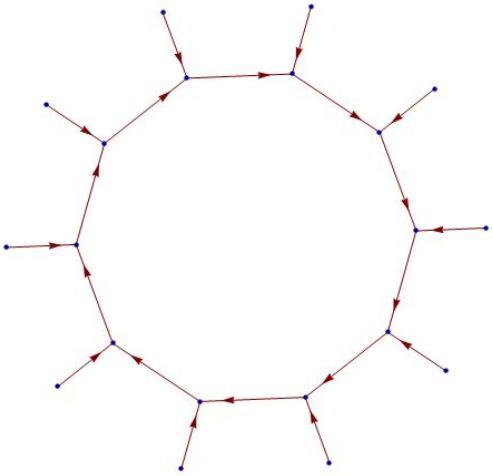

Abbildung 56: Visualisierung der MutexStar-Sequenz

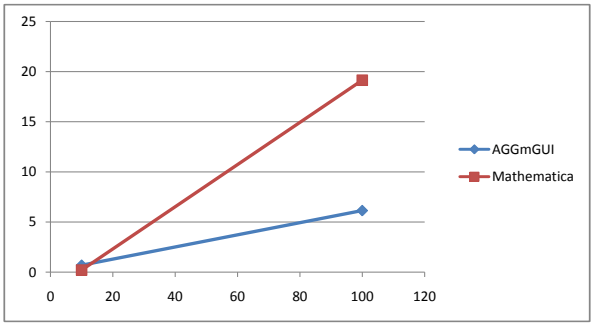

Abbildung 57: Ausführungszeiten Mutex

Die X-Achse des Diagrammes in Abbildung 57 beschreibt die Anzahl der Prozeßknoten und demnach den Komplexitätsgrad. An der Y-Achse stehen die Zeiteinheiten in Sekunden. Die Auswertung der Anwendungen bei einer Prozeßknotengröße von 10 ergab eine effizientere Berechnung unter Mathematica, während bei der Regelanwendung auf den Startgraphen mit 100 Prozeßknoten die AGG-Implementierung lediglich dreimal so schnell wie die Mathematica-Implementierung ist. Ein Großteil des Aufwandes bei der Realisierung der getypten attributiertn Graphtransformation bezieht sich ebenfalls auf die Matchsuche und auf das Überprüfen der negativen Anwendungsbedingungen.

Die Regelsequenz MutexStar wurde für die Prozeßknotengrößen 10 und 50 getestet. Eine Auswertung der Ausführungszeit für 10 Prozesse hat bei Mathematica einen durchschnittlichen Zeitwert von 1,138 Sekunden ergeben und liegt im Vergleich zu AGG mit 0,424 Sekunden nur geringfügig hinter AGG. Die Zeitwerte der Implementierung mit Mathematica bei einer Ausführung von 50 Prozessen sind ca. um den Faktor 10 langsamer als bei der AGG-Implementierung. Diese Differenzen sind jedoch u.a. auch durch das Ausführen der Matching-Funktionen bei Anwendungsbedingungen zu erklären. Das folgende Diagramm 58 zeigt die Ergebnisse der Zeitanalyse der MutexStar-Sequenz.

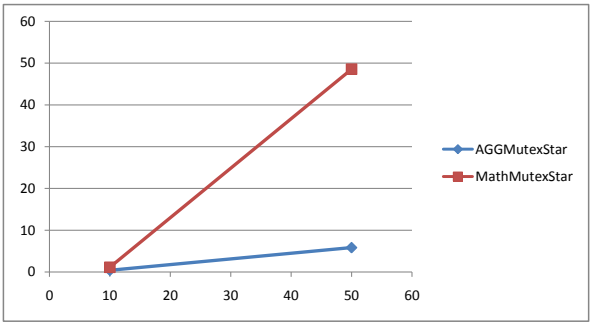

Abbildung 58: Ausführungszeiten MutexStar

Die Y-Achse beschreibt die Zeitwerte in Sekunden. Die X-Achse symbolisiert die Anzahl der Prozeßknoten.

8.3. AGG

Das an der TU Berlin entwickelte Tool AGG (Attributed Graph Grammar) [*AGG*] ist eine javabasierte Entwicklungsumgebung für algebraische Graphtransformationssysteme und eine visuelle Sprache. Die Implementierung erfolgte auf Basis der Idee der Colimiten und erlaubt die Anwendung von Regeln für attributierte Graphtransformationsysteme,

wobei das Verhalten des Systems durch die Graphtransformationsregeln spezifiziert wird. Komplexe Datenstrukturen werden in den Graphen gekapselt, die über einen dazugehörigen Typgraphen getypt sind.

Besonderheiten bei AGG sind Funktionalitäten wie die Kritische Paaranalyse und das Konsistenzchecking. Constraints, negative und positive Anwendungsbedingungen sind ebenfalls Teil der Anwendung. Eine AGG-Graphgrammatik kann durch die Analysetechniken der Kritischen Paaranalyse und des Konsistenzcheckings validiert werden.

Knoten und Kanten eines Graphen in AGG werden als Objekte bezeichnet, die wiederum Attribute enthalten können. Diese Attributierung erfolgt durch Java-Objekte und -Ausdrücke und ersetzt algebraische Spezifikationen und Terme. Basisdatentypen werden ebenfalls wie in Java zur Verfügung stehende Objektklassen verwendet. Regeln können Attributbedingungen in Form von aussagenlogischen Formeln besitzen.

Eine Graphgrammatik in AGG besteht aus einem Startgraphen und einer Menge von Regeln, die eventuell auch negative oder positive Anwendungsbedingungen enthalten können. In der GUI werden lediglich die linke und die rechte Regelseite visualisiert. Das Matching ist nicht auf injektive Matchings limitiert.

8.4. Weitere Graphtransformationstools

Derzeit konkurrieren zahlreiche Graphtransformationstools mit unterschiedlichen Stärken und Schwächen zueinander. Ich beschränken mich bei der direkten Vergleichsanalyse der Implementierung auf AGG. Im Artikel [*AGT*07] wurde AGG bereits mit einigen der hier erwähnten Tools in Bezug auf die Sierpinski-Implementierung verglichen.

Weitere Graphtransformationstools sind u.a. FUJABA, PROGRES, GrGen.NET, VIATRA, MOFLON und GROOVE. Fujaba ist ein Java-basiertes, durch die FUJABA Development Gruppe an der TU Paderborn entwickeltes Tool [*FUJ*] . Die Abkürzung resultiert aus der Beschreibung „From UML to Java And Back Again". FUJABA kombiniert Klassendiagramme der UML und UML-Verhaltensdiagramme, wobei hier der Schwerpunkt auf UML-Aktivitätendiagrammen und UML-Objektdiagrammen liegt.
UML-Aktivitätendiagramme dienen der operationalen Spezifikation von Methoden und bestehen aus speziellen Objektdiagrammen, deren Veränderung mit Hilfe der Theorie der Graphtransformation erfolgt.

GROOVE [*GRO*] steht für **G**raphs for **O**bject-**O**riented **VE**rification und gewährleistet eine formale Umsetzung der Modelltransformation, der dynamischen Semantik und die Möglichkeit beide Aspekte mit Hilfe einer automatischen Analyse bzw. des Model Checking der resultierenden Graphtransformationsysteme zu verifizieren.

Desweiteren existiert das Graphtransformationstool GrGen.NET (Graph Rewrite Ge-

nerator), eine intuitive und expressive, auf C#-basierte Spezifikationssprache, die in effizienter Art und Weise C#-Code aus deklarativen Graphersetzungsregelspezifikationen generiert (siehe [*GRG*]). Die Semantik ist aufgebaut auf dem DPO-Ansatz, wobei auch der DPO-Ansatz aus der Graphtransformationstheorie realisiert wurde. Die Generierung des algorithmischen Kernel von Anwendungen dient dazu, um in Graphen strukturierte Daten zu bearbeiten. GrGen.NET enthält für eine Debugging-Funktion eine interaktive Shell und für die Benutzerfreundlichkeit eine GUI.

Das Graphtransformationstool PROGRES[*UP*]/[*PROG*] steht für **PRO**grammed **G**raph **RE**writing **S**ystems und vereint die Konzepte einer Spezifikationssprache und einer integrierten Umgebung. PROGRES wurde an der RWTH Aachen entwickelt. Für die Transformation steht eine PROGRES-Spezifikation zur Verfügung. Kanten sind in PROGRES nur getypt, nicht jedoch attributiert. Eine Graphtransformationsregel in PROGRES besteht aus einem Pattern der linken Seite und aus einem Pattern der rechten Seite. Umfasst werden bei der Graphtransformation Attributbedingungen und Modifikationen von Attributwerten, Vor- und Nachbedingungen, eingebettete Regeln und die Ausgabe von Parametern.

Zur Überprüfung der Konsistenz, Vollständigkeit und Abhängigkeitsanforderungen von UML-basierten Systemen ist die Graphtransformationsumgebung VIATRA einsetzbar [*VIATRA*]. In Kapitel 9.1 wird auf eine durch Berücksichtigung des Pattern-Matching optimierte Variante der VIATRA-Implementierung eingegangen und die daraus resultierenden Effizienzsteigerungen konkretisiert.

Das an der Technischen Universität Darmstadt konzipierte MOFLON [*MO*] ist ein Java basiertes Tool zur Modellanalyse, Modelltransformation und Modellintegration für eine Vielzahl von Modellierungssprachen wie bspw. UML und für domain-spezifische Sprachen und vereint die Konzepte der Graphtransformation und der Triple-Graph-Grammatiken.

8.5. AGG versus Konkurrenz

Um das mit Mathematica entwickelte Programm in die Reihe der verschiedenen Tools einordnen zu können, möchte ich kurz auf einige Vergleichsergebnisse von AGG hinweisen. In [*AGT*07] wurde AGG mit zahlreichen, in Konkurrenz stehenden Graphtransformationsumgebungen anhand des Sierpinski-Benchmarks verglichen. GROOVE ist bei der Generationsstufe 6 ungefähr zehnmal schneller als die AGG-Implementierung. VIATRA liegt bei der Generationsstufe 6 in einem ähnlichen Bereich wie GROOVE, wobei bei zunehmender Anzahl an Generationen die Kurve einen vergleichbaren Steigerungsfaktor wie AGG vorzuweisen hat. Die Ergebnisse von FUJABA beginnen bei einer Generationsgröße von 7 und liegen hier bei unter 10 Millisekunden. Die Ausführungszeiten bei GrGen.NET liegen zwischen der VIATRA und der FUJABA-Kurve. Die in [*AGT*07] publizierten Ergebnisse werden durch die Abbildung 59 illustriert.

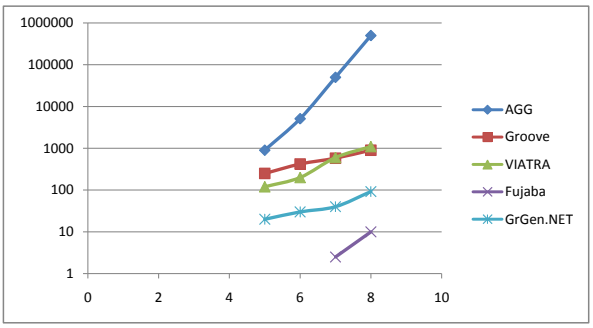

Abbildung 59: Ergebnisse Sierpinski

In Artikel [*BGT*05] werden zeitliche Ergebnisse des Mutex-Benchmarks von AGG, PROGRES und FUJABA gegenübergestellt. Auch hier zeigen sich Vorteile in den Implementierungen der PROGRES- und FUJABA-tools. In [*TR*07] werden ebenfalls die Ergebnisse der Ausführungszeiten der standardmäßigen Mutex-Variante und der Mutex-Star-Variante veröffentlicht. Das Diagramm 60 zeigt, daß in allen Stufen der Prozeßgrößen das Tool AGG langsamer ist als die in Konkurrenz stehenden Tools wie PROGRES, GrGEN und FUJABA.

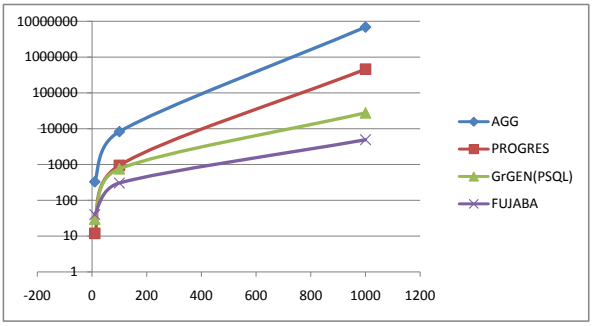

Abbildung 60: Ergebnisse Mutex

Angesichts der weitgefächerten Ansätze der Mathematicasoftware hinsichtlich möglicher Optimierungen, bestehen reelle Chancen, die Ausführungszeiten konkurrierender Graphtransforationsumgebungen zu erreichen.

Teil III.
Ausblick

9. Zukünftige Realisierungen

Weitere Ausbaustufen sind basierend auf der in Mathematica programmierten (getypten) attributierten Graphtransformation im Bereich der Triplegraphen und im Bereich der Graphtransformation mit Borrowed Context geplant. Im Vergleich zum Graphtransformationstool MOFLON (siehe 8.4) ist eine direkte formale Realisierung der Triple-Graph-Grammatiken geplant. Zudem ist eine Anbindung des Mathematica-Tools über die Schnittstellen MathLink und J/Link an mit Java implementierte Projekte im Zuge der Veranstaltungen „Visuelle Sprachen Projekt" an der TU Berlin ein erstrebenswertes Ziel.

Eine Optimierung der bisherigen Berechnungen der Graphtransformation kann u.a. durch ein Konzept des inkrementellen Pattern Matching erfolgen. Im folgenden Abschnitt 9.1 beschreibe ich diese Optimierungsmethode und orientiere mich an [*ICGT*08].

In der Vergleichsanalyse in Kapitel 8 wurde festgestellt, daß vor allem das Matching der Regelanwendungen den größten Aufwand verursacht. Eine weitere Optimierung des Matching würde eine weitere Verbesserung der Ausführungszeit mit sich bringen. Denkbar sind auch weitere Benchmark-Algorithmen wie bspw. Pacman, ein bereits bei AGG veröffentlichter Algorithmus als Testumgebung für die Graphtransformation.

9.1. Inkrementelles Pattern-Matching

Die Idee der musterbasierten Suche zur Identifizierung diskreter Strukturen oder Teilmengen durch vorgegebene Pattern findet in zahlreichen Programmiersprachen Verwendung. In [*ICGT*08] wird das inkrementelle Pattern-Matching im Bereich der Graphtransformation anhand zweier Beispiele erklärt. Eine Umsetzung der Implementierung ist in der Graphtransformationsumgebung VIATRA [*VIATRA*] erfolgt. Die Wiederverwendung von gespeicherten Informationen wird im Bereich der Petrinetze beim Schalten von Transitionen und bei der Modellsynchronisation genutzt. Bei der Modellsynchronisation werden Anpassungen in einem UML-Modell inkrementell an ein relationales Datenbankmodell weitergeleitet. Zugrunde liegt der RETE-Algorithmus, der auf Dr.Vharles L. Forgy zurückgeht und erstmalig im Jahre 1974 der Öffentlichkeit präsentiert wurde. In einem Netzwerk von Graphknoten erfolgt eine Speicherung partieller Matches eines Graph-Pattern. Ein partieller Match spezifiziert Modellelemente, die eine Teilmenge der Constraints erfüllen. Es erfolgt eine Einteilung in Wurzelknoten und Blattknoten des Graphen, wobei bis auf den Wurzelknoten alle Knoten mit einem Muster in der linken Regelseite korrespondieren. Der Weg eines Wurzelknotens zu einem Blattknoten bildet eine komplette linke Regelseite.

Um einen Einblick in die Optimierungsmöglichkeiten der Implementierung mit Hilfe des inkrementellen Pattern-Matching zu erhalten, möchte ich auf die Resultate der Vegleichs-

analyse von VIATRA/LS und VIATRA/RETE mit integriertem Pattern-Matching in [*ICGT*08] verweisen. Bezüglich der Ergebnisse der Synchronisation bei Petrinetzen kann man feststellen, daß bei einer Anzahl von 1000 Ausführungssequenzen die Kurve der optimierten Variante VIATRA/RETE bis zu einer Modellgröße von 10^5 linear verläuft und die Implementierung bereits bei einer Modellgröße von 100 um mehr als das zehnfache schneller ist. Eine deutliche Verbesserung zeigt sich desweiteren auch bei der Zeitanalyse der Synchronisation in Bezug auf das objekt-relationale Mapping. Bei einer Modellgröße von 500 ist die optimierte Implementierung ebenfalls um den Faktor 10 schneller.

Es stellt sich jedoch die Frage, ob eine Realisierung des inkrementellen Pattern-Matching für jegliche Beispieltransformationen relevant wäre. Bei diskreten Graphen ohne existierende Kanten oder bei injektiv-getypten Graphen gäbe es keine Verbesserung in der Ausführungszeit.

10. Zusammenfassung

Basierend auf den eingangs in Kapitel 5 angeführten theoretischen Konzepten der getypten attributierten Graphtransformation, wurde im darauffolgenden Kapitel 7 unter Berücksichtigung der besonderen Funktionalitäten der Software Mathematica eine effiziente Realisierung dieser Konzepte mit Mathematica vorgestellt. Neben der getypten attributierten Graphtransformation als zentraler Teil der Programmierung, wurde zusätzlich das theoretische Konzept der Pullback-Konstruktion als duales Gegenstück zur Pushout-Konstruktion implementiert. Die Pullback-Realisierung soll im weiteren Verlauf in Projekten im Bereich der Triple-Graph-Grammatiken integriert werden.

Anhand zweier für eine Testumgebung prädestinierter Benchmark-Beispiele erfolgte ein direkter Vergleich zur Graphtransformationsumgebung AGG. Trotz einer über viele Jahre gewachsenen und optimierten Programmierung im AGG-Projekt, liegen die Ergebnisse der Zeitanalyse der Mathematica-Implementierung im Bereich der AGG-Realisierung. In Kapitel 9 erfolgte ein kurzer Hinweis auf weitere optionale Erweiterungen und Verbesserungen, wobei insbesondere das inkrementelle Pattern-Matching als weitere Ausbaustufe geplant ist. Dieses Konzept ist explizit in Abschnitt 9.1 erläutert worden. Die außerordentlich dynamische Entwicklung der zugrunde liegenden Mathematica-Software kann hier durchaus als weiterer Vorteil angesehen werden.

Die Resultate der Benchmark-Beispiele Mutex und Sierpinski untermauern die Vorteile der Kombination aus einer Kompaktheit der Implementierung und einer hinsichtlich der Ausführungszeit der Regelanwendungen effizienten Umsetzung. In [*SCIB*09] ist zudem eine direkte Anwendung in der Praxis beschrieben.

Abschließend möchte ich auf ein Zitat aus dem Artikel [*SPON*] über die Mächtigkeit der Mathematica Software eingehen, die in ein neues Produkt namens Wolfram Alpha

eingebettet werden soll:

Vor fünfzig Jahren, in der Frühzeit der Computer, habe man erwartet, dass Maschinen bald mit all dem umgehen könnten, schreibt Wolfram: „Dass man dem Computer nur eine Sachfrage stellen müsste, und ihn die Antwort ausrechnen lässt. Aber dazu kam es nicht." Nun, glaube er, sei man an einem Punkt, an dem dieses Versprechen endlich einzulösen sei. Mit der Kombination aus Mathematica und seinen Theorien über basale, elementare Programme, die er „zelluläre Automaten" nennt.

Literatur

[AGG] AGG, Tool zur Simulation und Analyse von Graphgrammatiken. Entwickelt von TFS.
`http://tfs.cs.tu-berlin.de/agg`

[AGT07] G. Taentzer, E. Biermann, D. Bisztray, B. Bohnet, I. Boneva, A. Boronat, L. Geiger, R. Geiß, A. Horvath, O. Kniemeyer, T. Mens, B. Ness, D. PLump, T. Vajk, „Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools“, 2008.

[AGU] T. Modica, „Eine attributierte getypte Graphgrammatik zum syntaxgesteuerten Editieren von UML State Machines“, 2006.

[BGT05] G. Varró, A. Schürr, D. Varró, „Benchmarking for Graph Transformation“, 2005.

[BPMR08] G. Rangel, L. Lambers, B. Koenig, H. Ehrig, and Baldan P., „Behavior preservation on model refactoring using dpo transformations with borrowed contexts“, 2008.

[CAGT] J. Calleros, A. Stanciulescu, J. Vanderdonckt, J. Delacre, M. Winckler, „A Comparative Analysis of Graph Transformation Engines for User Interface Development“. Université catholique de Louvain, Université Toulouse, 2007.

[ECL] Eclipse, Open-Source Software-Entwicklungsumgebung.
`http://www.eclipse.org`

[FAGT] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, „Fundamentals of Algebraic Graph Transformation“. Springer, Berlin Heidelberg New York, 2005.

[FUJ] FUJABA, Graphtransformationstool.
`http://www.fujaba.de/`

[GRO] GROOVE, Graphtransformationstool.
`http://groove.sourceforge.net`

[GRG] GrGen, Graphtransformationstool.
`http://www.info.uni-karlsruhe.de/software/grgen/`

[ICGT08] G. Bergmann, A. Horvath, I. Rath, D. Varro, „A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation“. Budapest University of Technology and Economics, Department of Measurement and Information Systems, 1117 Budapest, Magyar Tud´osok krt. 2., 2008.

[MATH] Mathematica, Wolfram Research Homepage.
http://www.wolfram.com/

[MATL] MATLAB Homepage.
http://www.mathworks.de/

[MO] MOFLON Homepage.
http://www.moflon.org/index.html

[PROG] U. Ranger, E. Weinell, „The Graph Rewriting Language and Environment PROGRES“. Springer, Berlin Heidelberg, 2008.

[SCIB09] C. Brandt, F. Hermann, T. Engel, „Security and Consistency of IT and Business Models at Credit Suisse realized by Graph Constraints, Transformation and Integration using Algebraic Graph Theory“, 2009(To appear).

[SPON] Spiegel Online, „Software-Genie verspricht den Google-Killer “. Artikel vom 10.03.2009.
http://www.spiegel.de/netzwelt/tech/0,1518,612268,00.html

[TB07] G. Taentzer, E. Biermann, „Generating Sierpinski Triangles by the Tiger EMF Transformation Framework“. Philipps-Universität Marburg, Germany, Technische Universität Berlin, Germany, 2007.

[TR07] R. Geiss, M. Kroll, „On Improvements of the Varro Benchmark for Graph Transformation Tools“. Universität Karlsruhe(TH), 2007.

[UP] E. Weinell, U. Ranger, „Using PROGRES for Transforming UML Activity Diagrams into CSP Expressions“. RWTH Aachen University of Technology, Department of Computer Science 3 (Software Engineering), 2008.

[VIATRA] VIATRA Homepage.
http://www.eclipse.org/gmt/VIATRA2/

[WW] Wolfram Workbench.
http://www.additive-net.de/software/mathematica/workbench/index.shtml